

# Closure Representations on the .NET CLR

Experiments and  
Observations

Don Syme,  
Microsoft Research, Cambridge



## A real problem...



- .NET language-level interop is largely solved:
  - e.g. objects, calling conventions, exceptions
  - e.g. C# can easily call VB
  - e.g. SML can easily catch C# exceptions
  - e.g. SML.NET can understand C# classes
- BUT:
  - Scheme: Untyped function values
  - Standard ML: Typed function values
  - Haskell: Typed function values, also delayed computations
  - OCaml: Typed function values
  - Funnel: Typed function values, also delayed computations
  - Mercury: Typed function values
  - ...

## A real problem...



- No interop over ANY of these constructs
  - A function value from one language cannot easily be used in another language
  - There are subtle and not-so-subtle differences between them.
  - Even if there isn't, compilers all encode them in different ways.
- The question: Can we do anything about this?



- Definitions and Aims
- A Range of Possible Standardized Encodings
- Experimental Results
- Conclusions & Ways Forward

## Source Languages



# Source Languages



with some extra bits  
e.g. `(map2 (lambda (x y) (+ x y 2)) ...)`

generalize `(map2 (function foo) ...)`  
to `(map2 (lambda (x y) (+ x y 2)) ...)`  
using `(map2 (lambda (x y) (+ x y 2)) ...)`

Scheme

```
(map2 (function foo) ...)  
(def foo (z)  
  ... (map2 (lambda (x y) (+ x y z)) ...)
```

... (map2 (lambda (x y) (+ x y 2)) ...)

... (map2 (lambda (x y) (+ x y 2)) ...)  
... (map2 (lambda (x y) (+ x y 2)) ...)  
... (map2 (lambda (x y) (+ x y 2)) ...)

# Source Languages

Function passed as value

In Scheme args are passed  
as s-expressions

Scheme

```
(map2 (function foo) ...)  
(def foo (z)  
  ... (map2 (lambda (x y) (+ x y z)) ...)
```

Anonymous untyped functions

Scheme: implied  
context, not directly  
mutable

# Source Languages

Function passed as value

In Scheme args are passed  
as s-expressions

Scheme

```
(map2 (function foo) ...)  
(def foo (z)  
  ... (map2 (lambda (x y) (+ x y z)) ...)
```

Anonymous untyped functions

Scheme: implicit  
context, not directly  
mutable



# Source Languages

Function passed as value

In Scheme args are passed  
as s-expressions

Scheme

```
(map2 (function foo) ...)  
(def foo (z)  
  ... (map2 (lambda (x y) (+ x y z)) ...))
```

Anonymous untyped functions

Scheme: implied  
context, not directly  
mutable

ML

```
map2 (fn (x, y) => x + y + z) ...
```

A typed function expecting  
two integers, tupled

# Source Languages



Function passed as value

In Scheme args are passed  
as s-expressions

Scheme

```
(map2 (function foo) ...)  
(def foo (z)  
  ... (map2 (lambda (x y) (+ x y z)) ...))
```

Anonymous untyped functions

Scheme: implicit  
context, not directly  
mutable

ML

```
map2 (fn (x, y) => x + y + z) ...
```

A typed function expecting  
two integers, tupled

Again typed

OCaml

```
map2 (fun x y -> x + y + z) ...  
= map2 (fun x -> (fun y -> x + y + z)) ...
```

Functions return functions  
("currying") are very common

# Source Languages



Functions as *values* also

In Scheme args are passed  
as expressions

```
(map2 f x y) → f x y  
Rough (def foo (z)  
  (map2 ...)))
```

Anonymous untyped functions

scheme implicit  
context not directly  
relevant

ML `map2 : int → int → int`

Again typed

A typed function expecting  
two integers (typed)

```
ml  
-- map2 (fun x → (fun y → x + y + 2))
```

Haskell `map2 :: Int -> Int -> Int`

Functions return functions  
currying, are very common

Again typed, again lots of  
crazy, abstract, delayed  
computations

# Definitions



- Closures = functions in a context

context = Environment + Free variables

- Closure expressions = declarations that specify new inner functions and their contexts (context is usually implicit)
- Function value = code pointer + data = an object allocated to hold pointer(s) to the function code plus the environment
- Function application = invoking a function value
- Function type = a general class of types that function values belong to.

## Aims



- Fully-specified encoding of closures into MS-IL
- Easy to generate
- Supports separate compilation (no global analysis needed)
- Fast enough for e.g. OCaml
- Fits with MS-IL polymorphism/generics

## Example

70#



```
let table = new Table<int,string>(10);
let rename = 0;

let process update (valOld,valNew) =
    table.map (fun v -> if v = valOld then valNew else v)

let rename(valOld,valNew,v) =
    if (v = valOld) then
        rename++;
        valNew;
    else v;
```

Full Scala 3.0 page

## Example

Scala



A function object is passed in.

```
class Table<K,V> {  
  void map ( f: K => V ) {  
    {  
      for (int i = 0; ...) {  
        val' = f(val)  
      }  
    }  
    Pair<K,V>[] buckets;  
  }  
}
```

```
let table = new Table<int,string>(10).  
let rename = 0.
```

```
let process update (valOld,valNew) =  
  table.map (fun y => { if (y == valOld) valNew })
```

```
let rename(valOld,valNew,v) =  
  if (v == valOld) then  
    rename++;  
    valNew;  
  else v.
```

Function objects

# Example



//

A function `f` is passed in

```
class Table<K,V> {  
  void map ( int i, ... f )  
  {  
    for (int i = 0; ...) {  
      val' = f(val)  
    }  
  }  
  pair<K,V>[] buckets;  
}
```

Here the same function `f` is  
used on all buckets

```
let table = new Table<int,string>(10);  
let rename = 0;  
  
let process update (valOld,valNew) =  
  table.map ( ... on ... valNew ... )  
  
let rename(valOld,valNew,v) =  
  if (v == valOld) then  
    rename+;  
    valNew;  
  else v;
```

Functional language



# Example

GCM



1

A function object is passed in

```
class Table<K,V> {  
    void map ( int n, int m, f )  
    {  
        for (int i = 0; i < m; i++) {  
            val[i] = f(val[i])  
        }  
    }  
    Pair<K,V>[] buckets;  
}
```

Here the same function object is  
invoked n \* m times

Here we pass in a function  
that takes an old value and a  
new value and returns a new value

```
let table = new Table<int,string>(10);  
let rename = 0;
```

```
let process update (valOld,valNew) =  
    table.map (old,new,fun v => if (v == valOld) valNew v)
```

```
let rename(valOld,valNew,v) =  
    if (v == valOld) then  
        rename++;  
        valNew;  
    else v;
```

Functional language

# Example



Scala

A function object is passed in

```
class Table<K,V> {  
  void map ( f: K => V ) {  
    {  
      for (int i = 0; ...) {  
        val' = f(val)  
      }  
    }  
    Pair<K,V>[] buckets;  
  }  
}
```

Here the same function object is invoked many times

Here we pass in a function  
each time we call table.map, a  
new closure will be allocated

```
let table = new Table<Int,String>(10).  
let rename = 0.  
  
let process update (valOld,valNew) =  
  table.map { (v,v') => {  
    if (v == valOld) {  
      valNew  
    }  
    else {  
      v'  
    }  
  }  
}  
  
let rename(valOld,valNew,v) =  
  if (v == valOld) then  
    rename++;  
    valNew;  
  else v;
```

Function object is passed

That closure is generated each  
element from the environment  
Each time we call map a new  
function object will be allocated

# Example



//

A function object is passed in

```
class Table<K,V> {  
    void map (Function f) {  
        for (int i = 0; ...) {  
            val' = f(val)  
        }  
    }  
    Pair<K,V>[] buckets;  
}
```

Has the same function object  
invoked many times

Here we pass in a function  
each time we call table.map a  
new closure will be allocated

```
let table = new Table<int,string>(10);  
let rename = 0;  
  
let process update (valOld,valNew) =  
    table.map (val,v) => { new val = valNew }  
  
let rename(valOld,valNew,v) =  
    if (v == valOld) then  
        rename++;  
        valNew;  
    else v;
```

Function object

When a new function object  
calls this function, it can have  
side-effects

This closure expression takes  
elements from the environment  
each time we call it. A new  
function object will be allocated

# Example

GC #



```
class Table<K,V> {  
    void map ( System.Func<V,V> f )  
    {  
        for (int i = 0; ... ) {  
            val' = f(val);  
        }  
    }  
    Pair<K,V>[] buckets;  
}
```

```
let table = new Table<int,string>(10);  
let rename = 0;
```

```
let process update (valOld,valNew) =  
    table.map (v => (v, f(v, valNew)));
```

```
let (valOld, valNew, v) = rename (valOld, valNew);
```

```
let rename(valOld,valNew,v) =  
    if (v == valOld)  
        rename++;  
        valNew  
    else v
```

Future: Generalize to props

# Example

Scala



```
class Table<K,V> {  
  void map ( f: K => V ) {  
    {  
      for (int i = 0, ... ) {  
        val' = f(val)  
      }  
    }  
  }  
  List<K,V>[] buckets;  
}
```

A function object is added to

This is an extended generic function type

Here the same function object is  
invoked many times

Here we pass in a function  
Each time we call table.map a  
new closure will be allocated

```
let table = new Table<Int,String>(10).  
let rename = 0.  
  
let process update (valOld,valNew) =  
  table.map (v => { if (v == valOld) valNew })  
  
let rename(valOld,valNew,v) =  
  if (v == valOld) then  
    rename++;  
    valNew;  
  else v;
```

Function object image

When it creates the function object  
gives this function. It can have  
side-effects

This is the expected behavior  
elements from the environment  
Each time we call map, a new  
function object will be allocated

## Example



//

```
class Table<K,V> {  
    void map ( System.Func<V,V> f )  
    {  
        for (int i = 0; ...) {  
            val* = f(val);  
        }  
    }  
    Pair<K,V>[] buckets;  
}
```

```
let table = new Table<int,string>(10).  
let rename = 0.
```

```
let process update (valOld,valNew) =  
    table.map (new > x->valOld (if y==valNew) )
```

```
let f i j iOld iNew v = rename < i - j - iNew + iOld + 1
```

```
let rename(valOld,valNew,v) =  
    if (v == valOld)  
        rename++;  
        valNew  
    else v
```

Function f to process

# Example

RCM



```
class Table<K,V> {  
  void map ( System.Func<V,V> f )  
  {  
    for (int i = 0; ...) {  
      val' = f(val);  
    }  
  }  
  Pair<K,V>[] buckets;  
}
```

```
let table = new Table<int,string>(10).  
let rename = 0.
```

```
let process update (valOld,valNew) =  
  table.map (v => if (v == valOld) valNew else v)
```

```
let [valOld, valNew] = ...  
let rename = ...
```

```
let rename(valOld,valNew,v) =  
  if (v == valOld)  
    rename++;  
    valNew  
  else v
```

Function in place

Recursive if (v == valOld) ...  
are named with the environment  
violated

## Part 2: Implementation Choices





# Example

GC #



```
class Table<K,V> {  
    void map ( System.Func<V,V> f )  
    {  
        for (int i = 0; ...) {  
            vali = f(vali);  
        }  
    }  
    Pair<K,V>[] buckets;  
}
```

```
let table = new Table<int,string>(10).  
let rename = 0.
```

```
let process update (valOld, valNew) =  
    table.map (v => (v == valOld ? valNew : v))
```

```
let ... [ ... ] valNew v = rename < 1 ? v : new
```

```
let rename(valOld, valNew, v) =  
    if (v == valOld)  
        rename++;  
        valNew  
    else v
```

Function in place

Unlike `JS` for `JS` expressions  
are named with the environment  
isolated

## Part 2: Implementation Choices





- Note: for typed functional languages we really assume Generics have been added to the CLR

Generics == Parametric Polymorphism == Templates

## Choices & Non-choices



```
class Table<K,V> {  
  void map ( ... f ... )  
  {  
    for (int i = 0; ...) {  
      val* = f(val);  
    }  
  }  
  Pair<K,V>[] buckets;  
}
```

```
let table = new Table<int,string>(10).  
let rename = 0.  
  
let process_update (valOld,valNew) =  
  table.map (new clo[valOld,valNew]).  
  
let clo[valOld,valNew](y) = rename (valOld,valNew,y).  
  
let rename(valOld,valNew,v) =  
  if (v = valOld)  
    rename++;  
    valNew  
  else v
```

# Choices & Non-choices

Function & function types must  
be structural



```
class Table<K,V> {  
  void map (f: (V) => V) {  
    {  
      for (int i = 0; ...) {  
        val' = f(val);  
      }  
    }  
    pair<K,V>[] buckets;  
  }  
}
```

```
let table = new Table<int,string>(10).  
let rename = 0.  
  
let process update (valOld,valNew) =  
  table.map (new clo[valOld,valNew]).  
  
let clo[valOld,valNew](y) = rename (valOld,valNew.y);  
  
let rename(valOld,valNew,v) =  
  if (v = valOld)  
    rename+1,  
    valNew  
  else v
```

# Choices & Non-choices

Anonymous function types must  
be structural



```
class Table<K,V> {  
  void map ( (V => V) f ) {  
    {  
      for (int i = 0; ... ) {  
        val' = f(val);  
      }  
    }  
    Pair<K,V>[] buckets,  
  }
```

Encoding function types as  
generic  $M \rightarrow N$  types results  
in exact structural equivalence  
rules for free

Note  $\rightarrow$  and  $\lambda$  (and those that  
 $M \rightarrow N$  ) make delegates  
structural

Note  $\rightarrow$  and  $\lambda$  might still be  
a less able less useful in  
a generic  $\lambda$  - delegates

```
let table = new Table<int,string>(10);  
let rename = 0;
```

```
let process update (valOld,valNew) =  
  table.map (new clo[valOld,valNew]).
```

```
let clo[valOld,valNew](y) = rename (valOld,valNew,y);
```

```
let rename(valOld,valNew,v) =  
  if (v = valOld)  
    rename++,  
    valNew  
  else v
```

## Choices & Non-choices



Note: choice functions objects are  
heap-allocated

```
class Table<K,V> {  
  void map ( (V) -> (V) f )  
  {  
    for (int i = 0; ...) {  
      val' = f(val)  
    }  
  }  
  Pair<K,V>[] buckets;  
}
```

```
let table = new Table<int,string>(10).  
let rename = 0.  
  
let process update (valOld,valNew) =  
  table.map ( (v) -> (v + " (" + v + ", " + valNew + ")").  
  
let clojure (y) = 'New (y) = rename (valOld,valNew.y).  
  
let rename(valOld,valNew,v) =  
  if (v = valOld)  
    rename++;  
    valNew  
  else v
```

# Choices & Non-choices



Non-merge functions like ls are  
heap allocated

```
class Table<K,V> {  
  void map ( int m, int n, f )  
  {  
    for (int i = 0; ... ) {  
      val* = f(val)  
    }  
  }  
  Pair<K,V>[] buckets;  
}
```



```
let table = new Table<int,string>(10).  
let rename = 0.  
  
let process update (valOld,valNew) =  
  table.map (renam * (lambda (x) (valOld, valNew))).  
  
let clo|valOld|valNew (y) = rename (valOld,valNew,y);  
  
let rename(valOld,valNew,v) =  
  if (v = valOld)  
    rename++,  
    valNew  
  else v
```



# Choices & Non-choices



How to use functions: objects are heap-allocated

```
class Table<K,V> {  
  void map ( int i, int j, int k, int l, int m, int n, int o, int p )  
  {  
    for (int i = 0; ...) {  
      val' = f(val)  
    }  
  }  
  Pair<K,V>[] buckets;  
}
```



```
let table = new Table<int,string>(10).  
let rename = 0.
```

```
let process update (valOld,valNew) =  
  table.map ( ... (valOld, valNew) ).
```

```
let clo(x,y) = ... New (y) = rename (valOld,valNew,y)
```

```
let rename(valOld,valNew,v) =  
  if (v = valOld)  
    rename++,  
    valNew  
  else v
```

Choice: How is the environment stored w.r.t. the function object?

The relevant parts of the environment must be somehow accessed via fields of the function object

# Choices & Non-choices



here How do we get from the function value to the table? How is the code stored in a boxed heap-allocated function object?

getting from code B is an indirect call to the JIT CLR class tables interface tables delegates code pointers are the only choices

Non-choice fu  
heap-allocate

```
class Table<K,V> {  
    void map (int m, int n, T J)  
    {  
        for (int i = 0; ...) {  
            val' = f(val)  
        }  
    }  
    pair<K,V>[] buckets;  
}
```



```
let table = new Table<int,string>(10).  
let rename = 0.  
  
let process update (valOld,valNew) =  
    table.map (x,y -> (x,y + valNew)).  
  
let clo[valOld]: new (y) = rename (valOld,valNew.y)  
  
let rename(valOld,valNew,v) =  
    if (v = valOld)  
        rename++;  
        valNew  
    else v
```

here How is the environment stored w.r.t the function object?

the relevant parts of the environment must be somehow accessed via fields of the function object

# The Simplest Implementation



- Encoding into virtual methods

```
abstract class System.Func<A,B> {  
    virtual B apply(A);  
}  
  
class clo : System.Func<string,string> {  
    private string valOld;  
    private string valNew;  
    override string apply(string) { }  
}  
  
    new clo(...)  
    f.apply()
```

- Code = virtual method
- Environment stored inline in object

# The Simplest Implementation



- Encoding into virtual methods

This is the space of function types

```
abstract class system func<A,B> {  
    virtual B apply(A).  
}
```

Each closure has since a  
delegate class

```
class clo< system func<string,string> {  
    private string valold,  
    private string valnew  
    override string apply(string) {    }  
}
```

```
new clo(...)  
f.apply() ...
```

- Code = virtual method
- Environment stored inline in object

# The Simplest Implementation



- Encoding into virtual methods

This is the space of function types

```
abstract class System Func<A,B> {  
    virtual B apply(A),  
}
```

Each class becomes a delegate class

```
class clo : System Func<string,string> {  
    private string valold,  
    private string valnew  
    override string apply(string) {  
    }  
}
```

Closure construction makes an object

```
    new clo(...)  
    , f.apply() ... ~
```

Application becomes virtual method invocation.

- Code = virtual method
- Environment stored inline in object

# The Delegate Implementation



- Encoding into virtual methods

```
delegate B System.Func<A,R>(A)

class clo {
    private string valOld,
    private string valNew,
    string apply(string) {      }
}

new clo(...)  
new System.Func<string,string>(clo apply)  
  
f.Invoke("abc");
```

- Code = method on deleguee
- Environment stored in deleguee

# The Delegate Implementation



- Encoding into virtual methods

This is the space of function types

```
delegate B System.Func<A,B>(A)

class clo {
    private string valOld,
    private string valNew,
    string apply(string) {    }
}
```

Each closure becomes a delegate class

```
new clo(...)
new System.Func<string,string>(clo apply)

f.Invoke("abc");
```

- Code = method on delegatee
- Environment stored in delegatee

# The Delegate Implementation



- Encoding into virtual methods

This is the space of function types

```
delegate B System Func<A,B>(A);
```

```
class clo {  
    private string valOld;  
    private string valNew;  
    string apply(string) {  
    }  
}
```

Each closure becomes a delegate class

Closure construction makes a delegate and a delegate

```
new clo(...)  
new System.Func<string,string>(clo apply)  
  
f.Invoke("abc");
```

- Code = method on delegatee
- Environment stored in delegatee



# The Delegate Implementation



- Encoding into virtual methods

This is the space of function types

```
delegate B System Func<A B>(A)

class clo {
  private string valOld,
  private string valNew,
  string apply(string) {      }
}
```

Each closure becomes a delegate class

Closure construction makes a delegate and a delegate

```
new clo(...) ←
new System.Func<string,string>(clo apply)

↑ *invoke("abc"),
```

Application becomes delegate invocation

- Code = method on delegatee
- Environment stored in delegatee

# A Library Implementation



- Encode into:

- MS-IL C-style function pointers

- Closure "template" classes

```
abstract class System.Func<A,B> {  
    void * code;  
}  
  
static method B app<A,B>(System.Func<A,B>, A x)  
{ ... }  
  
class System.Clo<E,A,B> : System.Func<A,B> {  
    Clo(E env, method B *(E env,A x) code),  
    public E,  
}
```

# A Library Implementation



- Encode into:

MS-IL C-style function pointers

Closure "template" classes

This is the class of function types

```
abstract class System.Func<A,B> {  
    void * code;  
}  
  
static method B app<A,B>(System.Func<A,B>, A x)  
{ ... }  
  
class System.Clo<E,A,B> : System.Func<A,B> {  
    Clo(E env, method B ^ (E env,A x) code),  
    public E,  
}
```

# A Library Implementation



- Encode into:

MS-IL C-style function pointers

This is the class of function types

Closure "template" classes

```
abstract class System Func<A,B> {  
    void * code;  
}  
  
static method B app<A,B>(System Func<A,B>, A x)  
{ ... }  
  
class System Clo<A,B> : System Func<A,B> {  
    Clo(E env, Method B *(E env,A x) code),  
    public E,  
}
```

This is a helper method for application.

# A Library Implementation



- Encode into:

MS-IL C-style function pointers

This is the class of function types

Closure 'template' classes

```
abstract class System Func<A,B> {  
    void * code;  
}
```

This is a helper method for application.

```
static method B app<A,B>(System Func<A,B>, A x)  
{ ... }
```

This class is a template for all closures

```
class System Clo<A,B> : System Func<A,B> {  
    Clo(E env, Method B *(E env,A x) code),  
    public E,  
}
```

# A Library Implementation



- Encode into:

MS-IL C-style function pointers

This is the class of function types

Closure "template" classes

```
abstract class System Func<A,B> {  
    void * code;  
}
```

This is a helper method for application.

```
static method B app<A,B>(System Func<A,B>, A x)  
{ ... }
```

This class is a template for all closures

```
class System Clo<A,B> : System Func<A,B> {  
    Clo(E env, method B *(E env,A x) code),  
    public E,  
}
```

This is a C-function pointer type in MS-IL

# A Library Implementation



- Encode into:

MS-IL C-style function pointers

This is the class of function types

Closure "template" classes

```
abstract class System.Func<A,B> {  
    void * code;  
}
```

This is a helper method for application.

```
static method B app<A,B>(System.Func<A,B>, A x)  
{ ... }
```

This class is a template for all closures

```
class System.Clo<A,B> : System.Func<A,B> {  
    Clo(E env, Method B *(E env,A x) code),  
    public B,  
}
```

This is the environment

This is a C-function pointer type in MS-IL

# A Library Implementation



- Encode into:

MS-IL C-style function pointers

This is the class of function types

Closure "template" classes

```
abstract class System.Func<A,B> {  
    void * code;  
}
```

This is a helper method for application.

```
static method B app<A,B>(System.Func<A,B>, A x)  
{ ... }
```

This class is a template for all closures

```
class System.Closure<A,B> : System.Func<A,B> {  
    Closure(E env, Method B *(E env,A x) code),  
    public B,  
}
```

This is the environment

This is a C-function pointer type in MS-IL

next code is executable but the implementation of this module  
At the unsafe to use if needed  
this is the library is code  
again



## Part 3: Further Details of Implementation Choices



# Choices: Environments



```
class Table<K,V> {  
    void map ( ... ) {  
        for (int i = 0; ... ) {  
            val' = ...  
        }  
        Pair<K,V>[] buckets,  
    }  
}
```



[ ... ]



# Choices: Environments



```
class Table<K,V> {  
    void map ( Pair<K,V>... f )  
    {  
        for (int i = 0, ... ) {  
            val' = f(  
        }  
    }  
    Pair<K,V>[] buckets,  
}
```



Pair val [bucket]



displays split of class



add to class



# Choices: Environments

For closure templates, a flat effect can be achieved by using tupling e.g. instantiating E' with e.g. Pair int int - or Tuple4 - int int int int

```
class Table<K,V> {  
  void map ( int m1, int m2, int f )  
  {  
    for (int i = 0, ... ) {  
      val' = f ...  
    }  
  }  
  Pair<K,V>[] buckets,  
}
```



Flat via tupling



Address shared address



Call to class

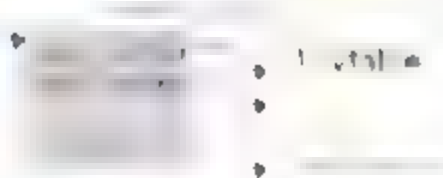


In our tests this is a new class per closure when needed

## Choices: Code



```
class Table<K,V> {  
    void map ( ... ) {  
        for (int i = 0, ... ) {  
            val' = ...  
        }  
        Pair<K,V>[] buckets,  
    }  
}
```



## Choices: Code



```
class Table<K,V> {  
    void map ( system function f(V,V) )  
    {  
        for (int i = 0, ... ) {  
            val' = f(  
        }  
    }  
    Pair<K,V>[] buckets,  
}
```



Practical Function  
Equation

1.  $v \mapsto v$

•

•

## Choices: Code



```
class Table<K,V> {  
    void map ( system::function< V const& (K const&)> f )  
    {  
        for (int i = 0; i < buckets; i++) {  
            V& val = buckets[i];  
            f(val);  
        }  
    }  
    Pair<K,V>[] buckets;  
}
```



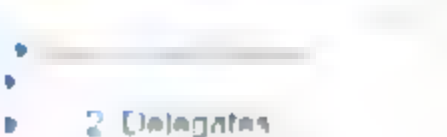
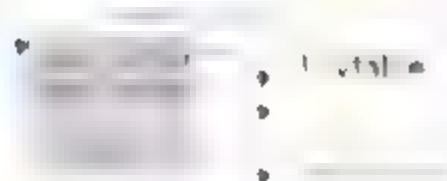
• `std::function`  
• `std::weak_ptr`

- 1 Vtable
- 
-

## Choices: Code



```
class Table<K,V> {  
    void map ( system func (V,V) P )  
    {  
        for (int i = 0 ; i < buckets ; i++) {  
            val' = P (val, val)  
        }  
    }  
    Pair<K,V>[] buckets,  
}
```



2 Delegates



# Choices: Code



```
class Table<K,V> {  
    void map ( ... ) {  
        for (int i = 0, ... ) {  
            val' = ...  
        }  
        Pair<K,V>[] buckets,  
    }  
}
```



→ ...  
→ ...  
→ ...



→ ...  
→ ...  
→ ...



→ ...  
→ ...

Code Pointers Own  
auto Temp. files

## Choices: Code

```
class Table<K,V> {  
    void map ( ... ) {  
        for (int i = 0, ... ) {  
            val' = f(  
        }  
    }  
    Pair<K,V>[] buckets,  
}
```

} Code Pointers Own  
auto templates



→ ...  
→ 1 Vtable  
→ ...  
→ ...



→ ...  
→ ...  
→ 1 delegates



→ ...  
→ ...

P, defining new code to update  
classes we get more control over code  
and environment layout



## Choices: Multiple Entries

### Schema

```
(map2 (lambda (x y) (f x y z)))
```

```
(def map2 (f l1 l2)  
  (f (car l1) (car l2)))
```

```
(def add2 (f x l2) .  
  (f x 4))
```

# Choices: Multiple Entries



## Scheme

```
(map2 (lambda (x y) (+ x y z)))
```

```
(def map2 (f l1 l2)
  (f (car l1) (car l2)))
```

```
(def add2 (f x l2) ...
  (f x 4))
```

optimized functional language  
in which arguments pass multiple entry point  
games

This also gives us  $\lambda$ -point as a function  
•  $\lambda$ -point  $\lambda$ -point is the need to allocate  
arguments as a  $\lambda$ -point

•  $\lambda$ -point  $\lambda$ -point  $\lambda$ -point  $\lambda$ -point is for  
passing 2 objects

This will then go much faster

# Choices: Multiple Entries



## Scheme

```
(map2 (lambda (x y) (+ x y 2))
```

```
(def map2 (f l1 l2)  
  (f (car l1) (car l2)))
```

optimized functional language  
no side effects, no GC, multiple entry point  
games

Does not pass any objects to a function  
expected to be fast. No need to allocate  
arguments as a Scheme

no other extra entry point or libraries for  
passing 2 objects

This will then go much faster

```
(def add2 (f x l2) .  
  (f x 4))
```

no other entry point e.g. objects

# Choices: Multiple Entries



## Scheme

```
(map2 (lambda (x y) (+ x y z)))
```

```
(def map2 (f l1 l2)  
  (f (car l1) (car l2)))
```

optimized functional language  
no side effects no ptrs, no objd entry point  
games

This is a ~~fast~~ ~~code~~ ~~type~~ ~~1~~ ~~to~~ ~~a~~ ~~function~~  
• ~~type~~ ~~1~~ ~~to~~ ~~2~~ ~~to~~ ~~3~~ ~~to~~ ~~4~~ ~~to~~ ~~5~~ ~~to~~ ~~6~~ ~~to~~ ~~7~~ ~~to~~ ~~8~~ ~~to~~ ~~9~~ ~~to~~ ~~10~~ ~~to~~ ~~11~~ ~~to~~ ~~12~~ ~~to~~ ~~13~~ ~~to~~ ~~14~~ ~~to~~ ~~15~~ ~~to~~ ~~16~~ ~~to~~ ~~17~~ ~~to~~ ~~18~~ ~~to~~ ~~19~~ ~~to~~ ~~20~~ ~~to~~ ~~21~~ ~~to~~ ~~22~~ ~~to~~ ~~23~~ ~~to~~ ~~24~~ ~~to~~ ~~25~~ ~~to~~ ~~26~~ ~~to~~ ~~27~~ ~~to~~ ~~28~~ ~~to~~ ~~29~~ ~~to~~ ~~30~~ ~~to~~ ~~31~~ ~~to~~ ~~32~~ ~~to~~ ~~33~~ ~~to~~ ~~34~~ ~~to~~ ~~35~~ ~~to~~ ~~36~~ ~~to~~ ~~37~~ ~~to~~ ~~38~~ ~~to~~ ~~39~~ ~~to~~ ~~40~~ ~~to~~ ~~41~~ ~~to~~ ~~42~~ ~~to~~ ~~43~~ ~~to~~ ~~44~~ ~~to~~ ~~45~~ ~~to~~ ~~46~~ ~~to~~ ~~47~~ ~~to~~ ~~48~~ ~~to~~ ~~49~~ ~~to~~ ~~50~~ ~~to~~ ~~51~~ ~~to~~ ~~52~~ ~~to~~ ~~53~~ ~~to~~ ~~54~~ ~~to~~ ~~55~~ ~~to~~ ~~56~~ ~~to~~ ~~57~~ ~~to~~ ~~58~~ ~~to~~ ~~59~~ ~~to~~ ~~60~~ ~~to~~ ~~61~~ ~~to~~ ~~62~~ ~~to~~ ~~63~~ ~~to~~ ~~64~~ ~~to~~ ~~65~~ ~~to~~ ~~66~~ ~~to~~ ~~67~~ ~~to~~ ~~68~~ ~~to~~ ~~69~~ ~~to~~ ~~70~~ ~~to~~ ~~71~~ ~~to~~ ~~72~~ ~~to~~ ~~73~~ ~~to~~ ~~74~~ ~~to~~ ~~75~~ ~~to~~ ~~76~~ ~~to~~ ~~77~~ ~~to~~ ~~78~~ ~~to~~ ~~79~~ ~~to~~ ~~80~~ ~~to~~ ~~81~~ ~~to~~ ~~82~~ ~~to~~ ~~83~~ ~~to~~ ~~84~~ ~~to~~ ~~85~~ ~~to~~ ~~86~~ ~~to~~ ~~87~~ ~~to~~ ~~88~~ ~~to~~ ~~89~~ ~~to~~ ~~90~~ ~~to~~ ~~91~~ ~~to~~ ~~92~~ ~~to~~ ~~93~~ ~~to~~ ~~94~~ ~~to~~ ~~95~~ ~~to~~ ~~96~~ ~~to~~ ~~97~~ ~~to~~ ~~98~~ ~~to~~ ~~99~~ ~~to~~ ~~100~~ ~~to~~ ~~101~~ ~~to~~ ~~102~~ ~~to~~ ~~103~~ ~~to~~ ~~104~~ ~~to~~ ~~105~~ ~~to~~ ~~106~~ ~~to~~ ~~107~~ ~~to~~ ~~108~~ ~~to~~ ~~109~~ ~~to~~ ~~110~~ ~~to~~ ~~111~~ ~~to~~ ~~112~~ ~~to~~ ~~113~~ ~~to~~ ~~114~~ ~~to~~ ~~115~~ ~~to~~ ~~116~~ ~~to~~ ~~117~~ ~~to~~ ~~118~~ ~~to~~ ~~119~~ ~~to~~ ~~120~~ ~~to~~ ~~121~~ ~~to~~ ~~122~~ ~~to~~ ~~123~~ ~~to~~ ~~124~~ ~~to~~ ~~125~~ ~~to~~ ~~126~~ ~~to~~ ~~127~~ ~~to~~ ~~128~~ ~~to~~ ~~129~~ ~~to~~ ~~130~~ ~~to~~ ~~131~~ ~~to~~ ~~132~~ ~~to~~ ~~133~~ ~~to~~ ~~134~~ ~~to~~ ~~135~~ ~~to~~ ~~136~~ ~~to~~ ~~137~~ ~~to~~ ~~138~~ ~~to~~ ~~139~~ ~~to~~ ~~140~~ ~~to~~ ~~141~~ ~~to~~ ~~142~~ ~~to~~ ~~143~~ ~~to~~ ~~144~~ ~~to~~ ~~145~~ ~~to~~ ~~146~~ ~~to~~ ~~147~~ ~~to~~ ~~148~~ ~~to~~ ~~149~~ ~~to~~ ~~150~~ ~~to~~ ~~151~~ ~~to~~ ~~152~~ ~~to~~ ~~153~~ ~~to~~ ~~154~~ ~~to~~ ~~155~~ ~~to~~ ~~156~~ ~~to~~ ~~157~~ ~~to~~ ~~158~~ ~~to~~ ~~159~~ ~~to~~ ~~160~~ ~~to~~ ~~161~~ ~~to~~ ~~162~~ ~~to~~ ~~163~~ ~~to~~ ~~164~~ ~~to~~ ~~165~~ ~~to~~ ~~166~~ ~~to~~ ~~167~~ ~~to~~ ~~168~~ ~~to~~ ~~169~~ ~~to~~ ~~170~~ ~~to~~ ~~171~~ ~~to~~ ~~172~~ ~~to~~ ~~173~~ ~~to~~ ~~174~~ ~~to~~ ~~175~~ ~~to~~ ~~176~~ ~~to~~ ~~177~~ ~~to~~ ~~178~~ ~~to~~ ~~179~~ ~~to~~ ~~180~~ ~~to~~ ~~181~~ ~~to~~ ~~182~~ ~~to~~ ~~183~~ ~~to~~ ~~184~~ ~~to~~ ~~185~~ ~~to~~ ~~186~~ ~~to~~ ~~187~~ ~~to~~ ~~188~~ ~~to~~ ~~189~~ ~~to~~ ~~190~~ ~~to~~ ~~191~~ ~~to~~ ~~192~~ ~~to~~ ~~193~~ ~~to~~ ~~194~~ ~~to~~ ~~195~~ ~~to~~ ~~196~~ ~~to~~ ~~197~~ ~~to~~ ~~198~~ ~~to~~ ~~199~~ ~~to~~ ~~200~~ ~~to~~ ~~201~~ ~~to~~ ~~202~~ ~~to~~ ~~203~~ ~~to~~ ~~204~~ ~~to~~ ~~205~~ ~~to~~ ~~206~~ ~~to~~ ~~207~~ ~~to~~ ~~208~~ ~~to~~ ~~209~~ ~~to~~ ~~210~~ ~~to~~ ~~211~~ ~~to~~ ~~212~~ ~~to~~ ~~213~~ ~~to~~ ~~214~~ ~~to~~ ~~215~~ ~~to~~ ~~216~~ ~~to~~ ~~217~~ ~~to~~ ~~218~~ ~~to~~ ~~219~~ ~~to~~ ~~220~~ ~~to~~ ~~221~~ ~~to~~ ~~222~~ ~~to~~ ~~223~~ ~~to~~ ~~224~~ ~~to~~ ~~225~~ ~~to~~ ~~226~~ ~~to~~ ~~227~~ ~~to~~ ~~228~~ ~~to~~ ~~229~~ ~~to~~ ~~230~~ ~~to~~ ~~231~~ ~~to~~ ~~232~~ ~~to~~ ~~233~~ ~~to~~ ~~234~~ ~~to~~ ~~235~~ ~~to~~ ~~236~~ ~~to~~ ~~237~~ ~~to~~ ~~238~~ ~~to~~ ~~239~~ ~~to~~ ~~240~~ ~~to~~ ~~241~~ ~~to~~ ~~242~~ ~~to~~ ~~243~~ ~~to~~ ~~244~~ ~~to~~ ~~245~~ ~~to~~ ~~246~~ ~~to~~ ~~247~~ ~~to~~ ~~248~~ ~~to~~ ~~249~~ ~~to~~ ~~250~~ ~~to~~ ~~251~~ ~~to~~ ~~252~~ ~~to~~ ~~253~~ ~~to~~ ~~254~~ ~~to~~ ~~255~~ ~~to~~ ~~256~~ ~~to~~ ~~257~~ ~~to~~ ~~258~~ ~~to~~ ~~259~~ ~~to~~ ~~260~~ ~~to~~ ~~261~~ ~~to~~ ~~262~~ ~~to~~ ~~263~~ ~~to~~ ~~264~~ ~~to~~ ~~265~~ ~~to~~ ~~266~~ ~~to~~ ~~267~~ ~~to~~ ~~268~~ ~~to~~ ~~269~~ ~~to~~ ~~270~~ ~~to~~ ~~271~~ ~~to~~ ~~272~~ ~~to~~ ~~273~~ ~~to~~ ~~274~~ ~~to~~ ~~275~~ ~~to~~ ~~276~~ ~~to~~ ~~277~~ ~~to~~ ~~278~~ ~~to~~ ~~279~~ ~~to~~ ~~280~~ ~~to~~ ~~281~~ ~~to~~ ~~282~~ ~~to~~ ~~283~~ ~~to~~ ~~284~~ ~~to~~ ~~285~~ ~~to~~ ~~286~~ ~~to~~ ~~287~~ ~~to~~ ~~288~~ ~~to~~ ~~289~~ ~~to~~ ~~290~~ ~~to~~ ~~291~~ ~~to~~ ~~292~~ ~~to~~ ~~293~~ ~~to~~ ~~294~~ ~~to~~ ~~295~~ ~~to~~ ~~296~~ ~~to~~ ~~297~~ ~~to~~ ~~298~~ ~~to~~ ~~299~~ ~~to~~ ~~300~~ ~~to~~ ~~301~~ ~~to~~ ~~302~~ ~~to~~ ~~303~~ ~~to~~ ~~304~~ ~~to~~ ~~305~~ ~~to~~ ~~306~~ ~~to~~ ~~307~~ ~~to~~ ~~308~~ ~~to~~ ~~309~~ ~~to~~ ~~310~~ ~~to~~ ~~311~~ ~~to~~ ~~312~~ ~~to~~ ~~313~~ ~~to~~ ~~314~~ ~~to~~ ~~315~~ ~~to~~ ~~316~~ ~~to~~ ~~317~~ ~~to~~ ~~318~~ ~~to~~ ~~319~~ ~~to~~ ~~320~~ ~~to~~ ~~321~~ ~~to~~ ~~322~~ ~~to~~ ~~323~~ ~~to~~ ~~324~~ ~~to~~ ~~325~~ ~~to~~ ~~326~~ ~~to~~ ~~327~~ ~~to~~ ~~328~~ ~~to~~ ~~329~~ ~~to~~ ~~330~~ ~~to~~ ~~331~~ ~~to~~ ~~332~~ ~~to~~ ~~333~~ ~~to~~ ~~334~~ ~~to~~ ~~335~~ ~~to~~ ~~336~~ ~~to~~ ~~337~~ ~~to~~ ~~338~~ ~~to~~ ~~339~~ ~~to~~ ~~340~~ ~~to~~ ~~341~~ ~~to~~ ~~342~~ ~~to~~ ~~343~~ ~~to~~ ~~344~~ ~~to~~ ~~345~~ ~~to~~ ~~346~~ ~~to~~ ~~347~~ ~~to~~ ~~348~~ ~~to~~ ~~349~~ ~~to~~ ~~350~~ ~~to~~ ~~351~~ ~~to~~ ~~352~~ ~~to~~ ~~353~~ ~~to~~ ~~354~~ ~~to~~ ~~355~~ ~~to~~ ~~356~~ ~~to~~ ~~357~~ ~~to~~ ~~358~~ ~~to~~ ~~359~~ ~~to~~ ~~360~~ ~~to~~ ~~361~~ ~~to~~ ~~362~~ ~~to~~ ~~363~~ ~~to~~ ~~364~~ ~~to~~ ~~365~~ ~~to~~ ~~366~~ ~~to~~ ~~367~~ ~~to~~ ~~368~~ ~~to~~ ~~369~~ ~~to~~ ~~370~~ ~~to~~ ~~371~~ ~~to~~ ~~372~~ ~~to~~ ~~373~~ ~~to~~ ~~374~~ ~~to~~ ~~375~~ ~~to~~ ~~376~~ ~~to~~ ~~377~~ ~~to~~ ~~378~~ ~~to~~ ~~379~~ ~~to~~ ~~380~~ ~~to~~ ~~381~~ ~~to~~ ~~382~~ ~~to~~ ~~383~~ ~~to~~ ~~384~~ ~~to~~ ~~385~~ ~~to~~ ~~386~~ ~~to~~ ~~387~~ ~~to~~ ~~388~~ ~~to~~ ~~389~~ ~~to~~ ~~390~~ ~~to~~ ~~391~~ ~~to~~ ~~392~~ ~~to~~ ~~393~~ ~~to~~ ~~394~~ ~~to~~ ~~395~~ ~~to~~ ~~396~~ ~~to~~ ~~397~~ ~~to~~ ~~398~~ ~~to~~ ~~399~~ ~~to~~ ~~400~~ ~~to~~ ~~401~~ ~~to~~ ~~402~~ ~~to~~ ~~403~~ ~~to~~ ~~404~~ ~~to~~ ~~405~~ ~~to~~ ~~406~~ ~~to~~ ~~407~~ ~~to~~ ~~408~~ ~~to~~ ~~409~~ ~~to~~ ~~410~~ ~~to~~ ~~411~~ ~~to~~ ~~412~~ ~~to~~ ~~413~~ ~~to~~ ~~414~~ ~~to~~ ~~415~~ ~~to~~ ~~416~~ ~~to~~ ~~417~~ ~~to~~ ~~418~~ ~~to~~ ~~419~~ ~~to~~ ~~420~~ ~~to~~ ~~421~~ ~~to~~ ~~422~~ ~~to~~ ~~423~~ ~~to~~ ~~424~~ ~~to~~ ~~425~~ ~~to~~ ~~426~~ ~~to~~ ~~427~~ ~~to~~ ~~428~~ ~~to~~ ~~429~~ ~~to~~ ~~430~~ ~~to~~ ~~431~~ ~~to~~ ~~432~~ ~~to~~ ~~433~~ ~~to~~ ~~434~~ ~~to~~ ~~435~~ ~~to~~ ~~436~~ ~~to~~ ~~437~~ ~~to~~ ~~438~~ ~~to~~ ~~439~~ ~~to~~ ~~440~~ ~~to~~ ~~441~~ ~~to~~ ~~442~~ ~~to~~ ~~443~~ ~~to~~ ~~444~~ ~~to~~ ~~445~~ ~~to~~ ~~446~~ ~~to~~ ~~447~~ ~~to~~ ~~448~~ ~~to~~ ~~449~~ ~~to~~ ~~450~~ ~~to~~ ~~451~~ ~~to~~ ~~452~~ ~~to~~ ~~453~~ ~~to~~ ~~454~~ ~~to~~ ~~455~~ ~~to~~ ~~456~~ ~~to~~ ~~457~~ ~~to~~ ~~458~~ ~~to~~ ~~459~~ ~~to~~ ~~460~~ ~~to~~ ~~461~~ ~~to~~ ~~462~~ ~~to~~ ~~463~~ ~~to~~ ~~464~~ ~~to~~ ~~465~~ ~~to~~ ~~466~~ ~~to~~ ~~467~~ ~~to~~ ~~468~~ ~~to~~ ~~469~~ ~~to~~ ~~470~~ ~~to~~ ~~471~~ ~~to~~ ~~472~~ ~~to~~ ~~473~~ ~~to~~ ~~474~~ ~~to~~ ~~475~~ ~~to~~ ~~476~~ ~~to~~ ~~477~~ ~~to~~ ~~478~~ ~~to~~ ~~479~~ ~~to~~ ~~480~~ ~~to~~ ~~481~~ ~~to~~ ~~482~~ ~~to~~ ~~483~~ ~~to~~ ~~484~~ ~~to~~ ~~485~~ ~~to~~ ~~486~~ ~~to~~ ~~487~~ ~~to~~ ~~488~~ ~~to~~ ~~489~~ ~~to~~ ~~490~~ ~~to~~ ~~491~~ ~~to~~ ~~492~~ ~~to~~ ~~493~~ ~~to~~ ~~494~~ ~~to~~ ~~495~~ ~~to~~ ~~496~~ ~~to~~ ~~497~~ ~~to~~ ~~498~~ ~~to~~ ~~499~~ ~~to~~ ~~500~~ ~~to~~ ~~501~~ ~~to~~ ~~502~~ ~~to~~ ~~503~~ ~~to~~ ~~504~~ ~~to~~ ~~505~~ ~~to~~ ~~506~~ ~~to~~ ~~507~~ ~~to~~ ~~508~~ ~~to~~ ~~509~~ ~~to~~ ~~510~~ ~~to~~ ~~511~~ ~~to~~ ~~512~~ ~~to~~ ~~513~~ ~~to~~ ~~514~~ ~~to~~ ~~515~~ ~~to~~ ~~516~~ ~~to~~ ~~517~~ ~~to~~ ~~518~~ ~~to~~ ~~519~~ ~~to~~ ~~520~~ ~~to~~ ~~521~~ ~~to~~ ~~522~~ ~~to~~ ~~523~~ ~~to~~ ~~524~~ ~~to~~ ~~525~~ ~~to~~ ~~526~~ ~~to~~ ~~527~~ ~~to~~ ~~528~~ ~~to~~ ~~529~~ ~~to~~ ~~530~~ ~~to~~ ~~531~~ ~~to~~ ~~532~~ ~~to~~ ~~533~~ ~~to~~ ~~534~~ ~~to~~ ~~535~~ ~~to~~ ~~536~~ ~~to~~ ~~537~~ ~~to~~ ~~538~~ ~~to~~ ~~539~~ ~~to~~ ~~540~~ ~~to~~ ~~541~~ ~~to~~ ~~542~~ ~~to~~ ~~543~~ ~~to~~ ~~544~~ ~~to~~ ~~545~~ ~~to~~ ~~546~~ ~~to~~ ~~547~~ ~~to~~ ~~548~~ ~~to~~ ~~549~~ ~~to~~ ~~550~~ ~~to~~ ~~551~~ ~~to~~ ~~552~~ ~~to~~ ~~553~~ ~~to~~ ~~554~~ ~~to~~ ~~555~~ ~~to~~ ~~556~~ ~~to~~ ~~557~~ ~~to~~ ~~558~~ ~~to~~ ~~559~~ ~~to~~ ~~560~~ ~~to~~ ~~561~~ ~~to~~ ~~562~~ ~~to~~ ~~563~~ ~~to~~ ~~564~~ ~~to~~ ~~565~~ ~~to~~ ~~566~~ ~~to~~ ~~567~~ ~~to~~ ~~568~~ ~~to~~ ~~569~~ ~~to~~ ~~570~~ ~~to~~ ~~571~~ ~~to~~ ~~572~~ ~~to~~ ~~573~~ ~~to~~ ~~574~~ ~~to~~ ~~575~~ ~~to~~ ~~576~~ ~~to~~ ~~577~~ ~~to~~ ~~578~~ ~~to~~ ~~579~~ ~~to~~ ~~580~~ ~~to~~ ~~581~~ ~~to~~ ~~582~~ ~~to~~ ~~583~~ ~~to~~ ~~584~~ ~~to~~ ~~585~~ ~~to~~ ~~586~~ ~~to~~ ~~587~~ ~~to~~ ~~588~~ ~~to~~ ~~589~~ ~~to~~ ~~590~~ ~~to~~ ~~591~~ ~~to~~ ~~592~~ ~~to~~ ~~593~~ ~~to~~ ~~594~~ ~~to~~ ~~595~~ ~~to~~ ~~596~~ ~~to~~ ~~597~~ ~~to~~ ~~598~~ ~~to~~ ~~599~~ ~~to~~ ~~600~~ ~~to~~ ~~601~~ ~~to~~ ~~602~~ ~~to~~ ~~603~~ ~~to~~ ~~604~~ ~~to~~ ~~605~~ ~~to~~ ~~606~~ ~~to~~ ~~607~~ ~~to~~ ~~608~~ ~~to~~ ~~609~~ ~~to~~ ~~610~~ ~~to~~ ~~611~~ ~~to~~ ~~612~~ ~~to~~ ~~613~~ ~~to~~ ~~614~~ ~~to~~ ~~615~~ ~~to~~ ~~616~~ ~~to~~ ~~617~~ ~~to~~ ~~618~~ ~~to~~ ~~619~~ ~~to~~ ~~620~~ ~~to~~ ~~621~~ ~~to~~ ~~622~~ ~~to~~ ~~623~~ ~~to~~ ~~624~~ ~~to~~ ~~625~~ ~~to~~ ~~626~~ ~~to~~ ~~627~~ ~~to~~ ~~628~~ ~~to~~ ~~629~~ ~~to~~ ~~630~~ ~~to~~ ~~631~~ ~~to~~ ~~632~~ ~~to~~ ~~633~~ ~~to~~ ~~634~~ ~~to~~ ~~635~~ ~~to~~ ~~636~~ ~~to~~ ~~637~~ ~~to~~ ~~638~~ ~~to~~ ~~639~~ ~~to~~ ~~640~~ ~~to~~ ~~641~~ ~~to~~ ~~642~~ ~~to~~ ~~643~~ ~~to~~ ~~644~~ ~~to~~ ~~645~~ ~~to~~ ~~646~~ ~~to~~ ~~647~~ ~~to~~ ~~648~~ ~~to~~ ~~649~~ ~~to~~ ~~650~~ ~~to~~ ~~651~~ ~~to~~ ~~652~~ ~~to~~ ~~653~~ ~~to~~ ~~654~~ ~~to~~ ~~655~~ ~~to~~ ~~656~~ ~~to~~ ~~657~~ ~~to~~ ~~658~~ ~~to~~ ~~659~~ ~~to~~ ~~660~~ ~~to~~ ~~661~~ ~~to~~ ~~662~~ ~~to~~ ~~663~~ ~~to~~ ~~664~~ ~~to~~ ~~665~~ ~~to~~ ~~666~~ ~~to~~ ~~667~~ ~~to~~ ~~668~~ ~~to~~ ~~669~~ ~~to~~ ~~670~~ ~~to~~ ~~671~~ ~~to~~ ~~672~~ ~~to~~ ~~673~~ ~~to~~ ~~674~~ ~~to~~ ~~675~~ ~~to~~ ~~676~~ ~~to~~ ~~677~~ ~~to~~ ~~678~~ ~~to~~ ~~679~~ ~~to~~ ~~680~~ ~~to~~ ~~681~~ ~~to~~ ~~682~~ ~~to~~ ~~683~~ ~~to~~ ~~684~~ ~~to~~ ~~685~~ ~~to~~ ~~686~~ ~~to~~ ~~687~~ ~~to~~ ~~688~~ ~~to~~ ~~689~~ ~~to~~ ~~690~~ ~~to~~ ~~691~~ ~~to~~ ~~692~~ ~~to~~ ~~693~~ ~~to~~ ~~694~~ ~~to~~ ~~695~~ ~~to~~ ~~696~~ ~~to~~ ~~697~~ ~~to~~ ~~698~~ ~~to~~ ~~699~~ ~~to~~ ~~700~~ ~~to~~ ~~701~~ ~~to~~ ~~702~~ ~~to~~ ~~703~~ ~~to~~ ~~704~~ ~~to~~ ~~705~~ ~~to~~ ~~706~~ ~~to~~ ~~707~~ ~~to~~ ~~708~~ ~~to~~ ~~709~~ ~~to~~ ~~710~~ ~~to~~ ~~711~~ ~~to~~ ~~712~~ ~~to~~ ~~713~~ ~~to~~ ~~714~~ ~~to~~ ~~715~~ ~~to~~ ~~716~~ ~~to~~ ~~717~~ ~~to~~ ~~718~~ ~~to~~ ~~719~~ ~~to~~ ~~720~~ ~~to~~ ~~721~~ ~~to~~ ~~722~~ ~~to~~ ~~723~~ ~~to~~ ~~724~~ ~~to~~ ~~725~~ ~~to~~ ~~726~~ ~~to~~ ~~727~~ ~~to~~ ~~728~~ ~~to~~ ~~729~~ ~~to~~ ~~730~~ ~~to~~ ~~731~~ ~~to~~ ~~732~~ ~~to~~ ~~733~~ ~~to~~ ~~734~~ ~~to~~ ~~735~~ ~~to~~ ~~736~~ ~~to~~ ~~737~~ ~~to~~ ~~738~~ ~~to~~ ~~739~~ ~~to~~ ~~740~~ ~~to~~ ~~741~~ ~~to~~ ~~742~~ ~~to~~ ~~743~~ ~~to~~ ~~744~~ ~~to~~ ~~745~~ ~~to~~ ~~746~~ ~~to~~ ~~747~~ ~~to~~ ~~748~~ ~~to~~ ~~749~~ ~~to~~ ~~750~~ ~~to~~ ~~751~~ ~~to~~ ~~752~~ ~~to~~ ~~753~~ ~~to~~ ~~754~~ ~~to~~ ~~755~~ ~~to~~ ~~756~~ ~~to~~ ~~757~~ ~~to~~ ~~758~~ ~~to~~ ~~759~~

## Part 4: Experimental Results



# ILX Implementations by Translation



•  $M \rightarrow_{\text{ILX}} M'$

$\vdash M \rightarrow_{\text{ILX}} M'$

$\vdash M \rightarrow_{\text{ILX}} M'$

$M \rightarrow_{\text{ILX}} M'$

$M \rightarrow_{\text{ILX}} M'$

•  $M \rightarrow_{\text{ILX}} M'$   
•  $M \rightarrow_{\text{ILX}} M'$   
•  $M \rightarrow_{\text{ILX}} M'$



# Simulated Performance Load



- Generated 100,000,000 calls to the `test` method of a typical class and program

```
Method invoked take a parameter  
then take  
and take
```

```
Method invoked by a class or object  
Method invoked by a class or object  
Method invoked by a class or object
```

All values are reported

Each class used contains a varying number of function calls

Each function also called a varying number of times

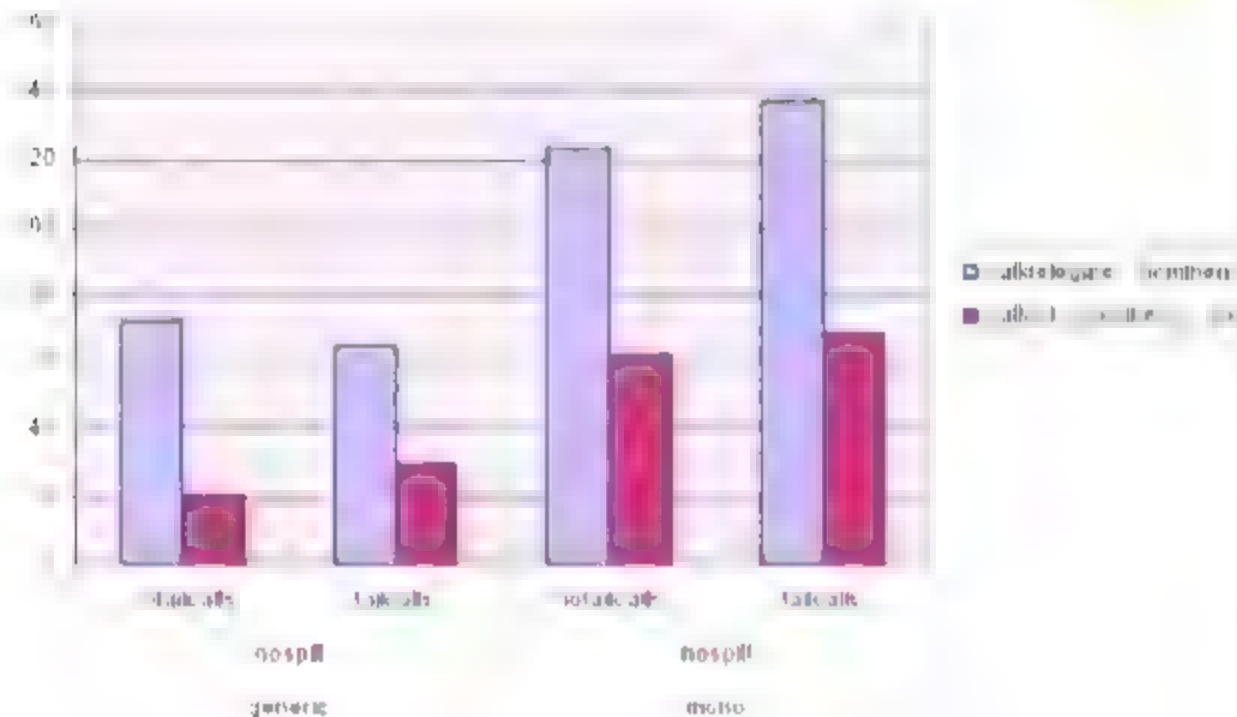
- BEST TIME ~ 11s





many easy targets

complicated targets





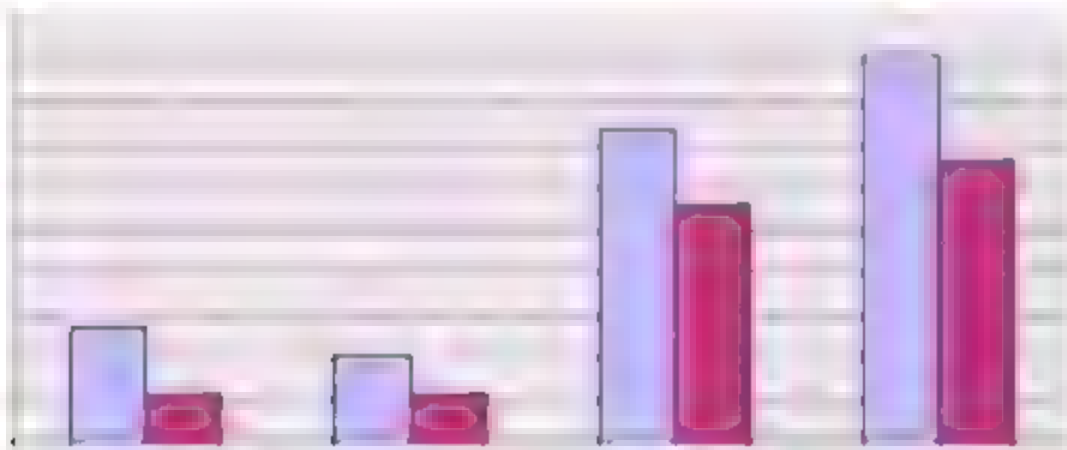


10. 10월 10일 10월 10일 10월 10일 10월 10일 10월 10일 10월 10일 10월 10일



10월 10일 10월 10일 10월 10일 10월 10일 10월 10일 10월 10일 10월 10일 10월 10일 10월 10일

Effect of teaching and development of science in a form development science

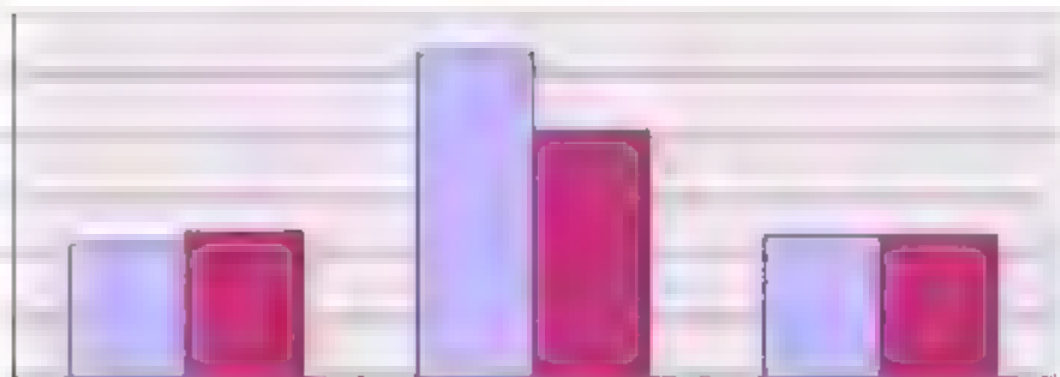


Control

Experimental



Figure 1: Using PLS and the Top-Types method of Model factors



■ PLS  
■ Top-Types



Figure 1: Using Plot() and as.factor() to convert categorical data to factors



color

type

red

blue

color

type

red

blue

color

type

red

blue

color

type

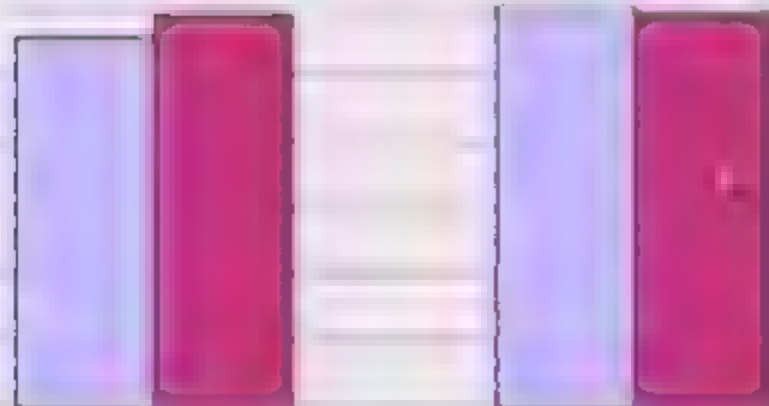
red

blue





Effect of Methyl B-Glucuronosyl Phosphonate on Core Template

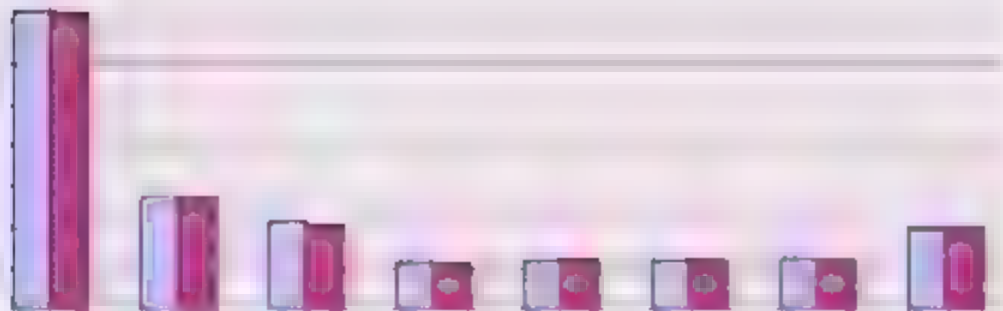


■

■



## Figure 1. Mean (SD) age at onset of symptoms



12

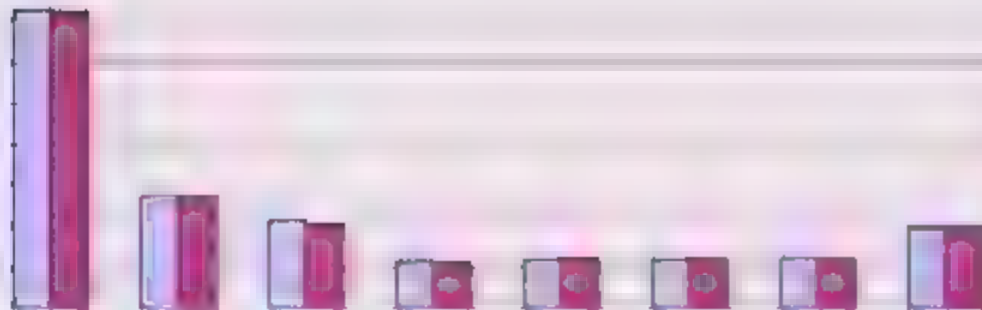
13

14

15



## er-fichte, Kiefer-fichte e -ent ude



11



# Conclusions



- 1 V1 Delegates suck
- 2 V1 Tailcalls suck
- 3 Generics are an extremely useful compilation device
- 4 Multiple entry points are feasible and worth it
- 5 Flat closures are feasible and worth it
- 6 Multiple environment slots are not worth it
- 7 A library-based approach is feasible

# Conclusions



- 1 V1 Delegates suck
- 2 V1 Tailcalls suck
- 3 Generics are an extremely useful compilation device
- 4 Multiple entry points are feasible and worth it
- 5 Flat closures are feasible and worth it
- 6 Multiple environment slots are not worth it
- 7 A library-based approach is feasible



# Related Concepts

- **Classes/Object**

- context is explicit, the fields

- certain mechanism to multiple method is static method is reflection

- **Java Method Class** `java.lang.reflect.Method`

- context is explicit

- associated with many methods

- **C# Delegates**

- Context is explicit

- Code and data are tied

- Also, explicit memory is not available

- **Method pointer**

- type is pointer to method

- • **Unboxed Delegate**

- Full features are much more useful

# Looking Forward



## Ways ahead

- 1. The compiler, it will depend on the set of programs plus a large set of abstractions

- delegates

• 1 delegates  
+  
Environment, symbols  
+  
[ Objects, list, set, if, methods,  
+  
Environments, name in delegate  
+  
Multiple envs, frames



# Summary



## Problems

1. Finding a common best practice and multiple type semantics
2. The most efficient standardized encoding shared library of types is to compare

## Future research

1. The capability to estimate performance of a series
2. Perhaps propose standard on things for other constructs

Questions?



# Runtime Verification and .NET

Michael Barnett

Wolfram Schulte

Foundations of Software Engineering

# Testing and Verification?

"Computer Science is akin to astronomy and astrology but it lacks the precision of the former and the popularity of the latter."

Specifications are Good...

---

## Specifications are Good ..

- So why doesn't anyone write them?

# Specifications are Good...

So why doesn't anyone write them?

- . Limited usefulness    dead documents

# Specifications are Good ..

- So why doesn't anyone write them?
    - . Limited usefulness dead documents
  - But some people **do** write them
-



# Specifications are Good...

So why doesn't anyone write them?

- Limited usefulness   dead documents

**But some people *do* write them**

- Programmers write assertions

# Specifications are Good ..

So why doesn't anyone write them?

- . Limited usefulness dead documents

**But some people do write them**

- . Programmers write assertions
- . Programmers write error checking

# Specifications are Good...

**So why doesn't anyone write them?**

- . Limited usefulness dead documents

**But some people *do* write them**

- . Programmers write assertions
- . Programmers write error checking
- . Programmers write exception handling

# Specifications are Good ..

**So why doesn't anyone write them?**

- . Limited usefulness dead documents

**But some people do write them**

- . Programmers write assertions
- . Programmers write error checking
- . Programmers write exception handling
- . Programmers write code'

# It's the level, stupid. .

- Specifications would be more useful if they were:
    - At a higher level
    - Available separate from source code
    - Enforceable
-

It's the level, stupid. .

Specifications would be more useful if they were:

- . At a higher level
- . Available separate from source code
- . Enforceable

i e., executable!

# The Big Picture

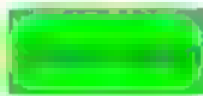
## Programming



Interface  
↓  
Implementation



## Specification and Verification



Constraints  
↓  
Implementation

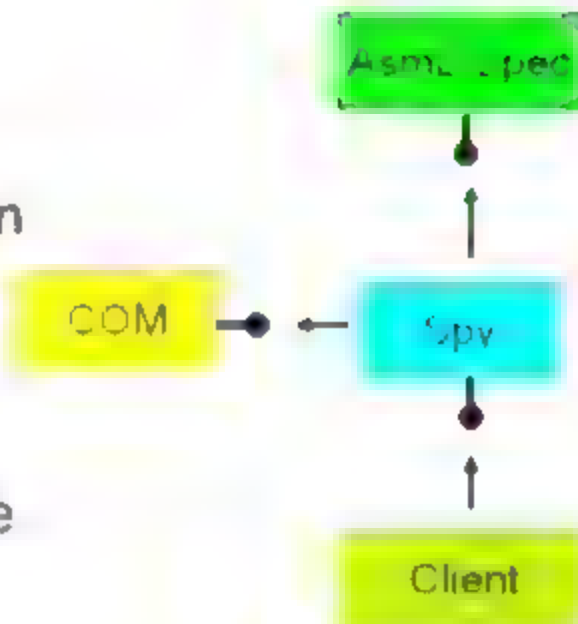
Proof of  
↓  
Implementation



Model  
↓  
Verification

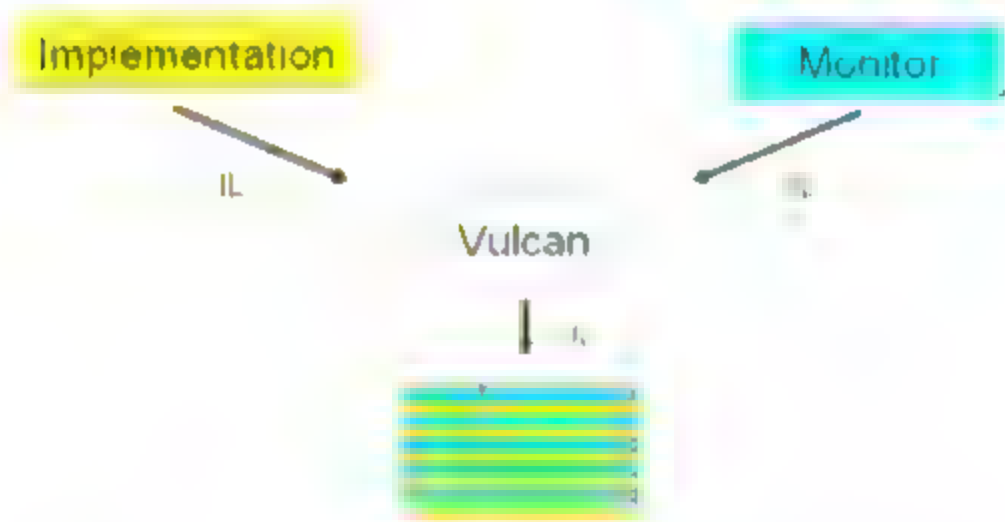
# Previously

- Observational equivalence
- No instrumentation required
- Non-determinism
- Callbacks
- State equivalence





# Tighter Integration: NET



# (Trivial) Example

```
interface IMath {  
    double sqrt(double x)  
}
```

```
class CMath implements IMath {  
    double sqrt(double x)  
    {  
        |  
        |  
        |  
    }  
}
```

```
class IMathContract implements IMath {  
    bool sqrt_pre(double x)  
    { x >= 0; }  
    bool sqrt_post(double x)  
    { result <= x; }  
}
```

# Monitored Implementation

```
class Math {  
    double sqrt(double x)  
    { double result  
      assert(x >= 0),  
      modified body  
      assert(result * result == x)  
      return result  
    }  
}
```

# Monitor Language

```
class Contract {  
    bool f_precond() {}  
    bool f_postcond() {}  
    bool f_classinvariant() {}  
    Object f_exception() {}  
}
```

# Monitor Language

```
class Contract {  
    bool f_precond() {}  
    bool f_postcond() {}  
    bool f_classinvariant() {}  
    Object f_exception() {}  
}
```

# What to specify?

## Properties

- . Pre-/post-conditions

## Calling protocols

- . Required method calls *mandatory calls*

## Model State

- . Required persistence *objects*

## Exceptions

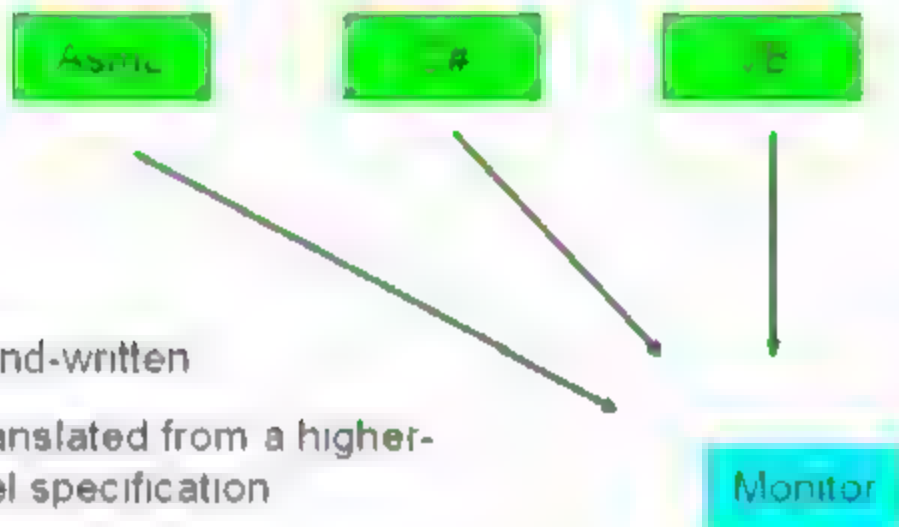
## (Trivial) Example

```
interface IMath {  
    double sqrt(double x);  
}
```

```
class CMath implements IMath {  
    double sqrt(double x)  
    {  
        ...  
    }  
}
```

```
class IMathContract { IMath {  
    bool sqrt_pre(double x)  
    { x >= 0; }  
    bool sqrt_post(double x)  
    { result >= x; }  
}
```

# Monitors





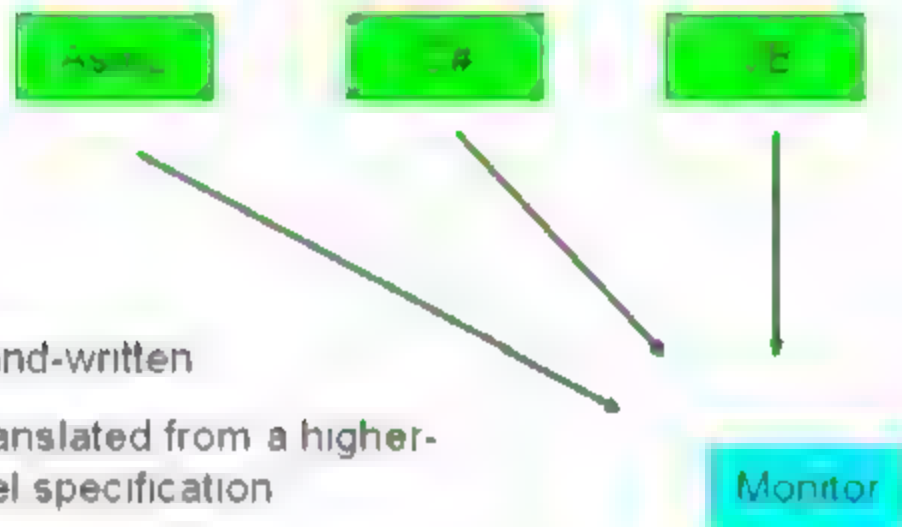
## (Trivial) Example

```
interface IMath {  
    double sqrt(double x);  
}
```

```
class CMath implements IMath {  
    double sqrt(double x)  
    {  
        |  
    }  
}
```

```
class IMathContract implements IMath {  
    bool sqrt_pre(double x)  
    {  
        [ x >= 0; ]  
    }  
    bool sqrt_post(double x)  
    {  
        result = result - x;  
    }  
}
```

# Monitors

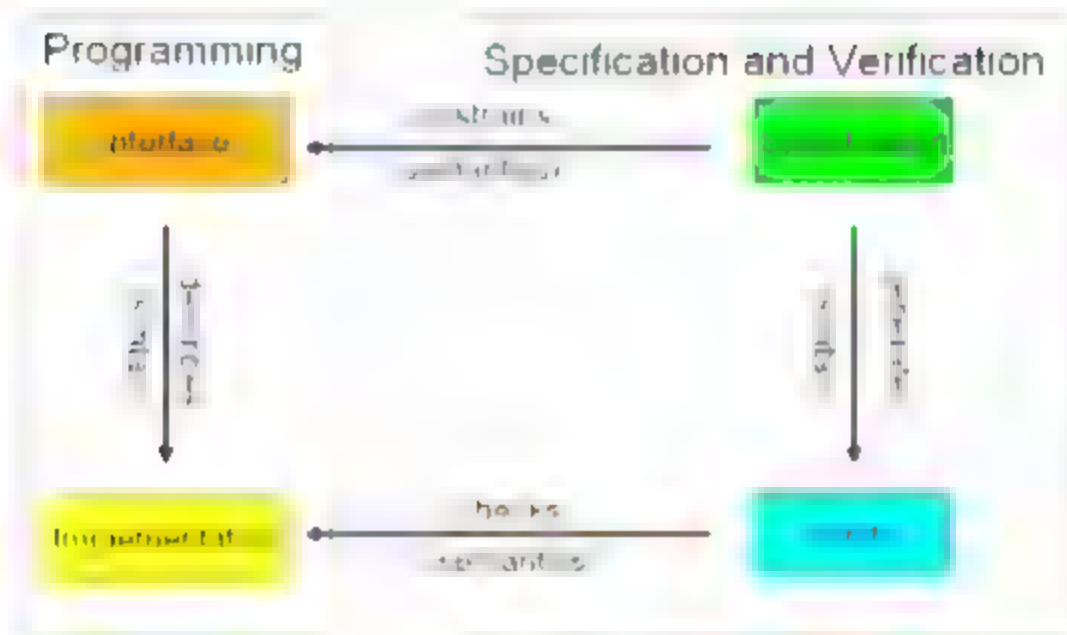


# AsmL. Microsoft's ASM-Language

## A powerful specification language:

- . Based on ASMs
- . Readable executable testable
- . Combines mathematical, object-oriented component-oriented approaches
- . Integrated into VS6 (working on VS NET)
- . Literate style Word and XML

# The Big Picture

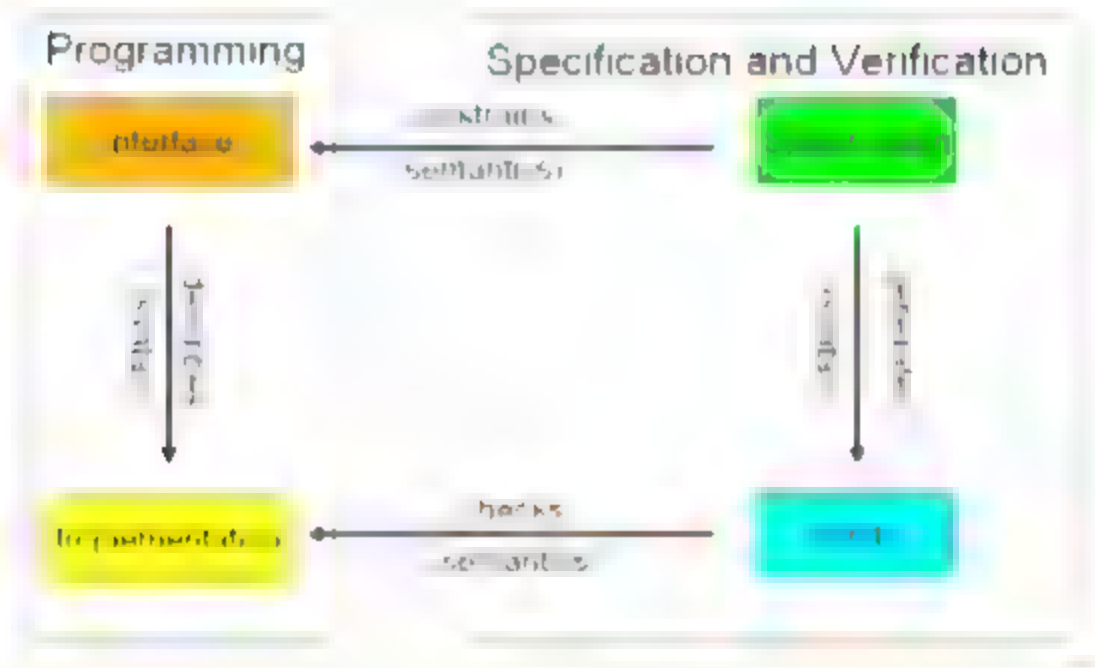


# AsmL Microsoft's ASM-Language

## A powerful specification language:

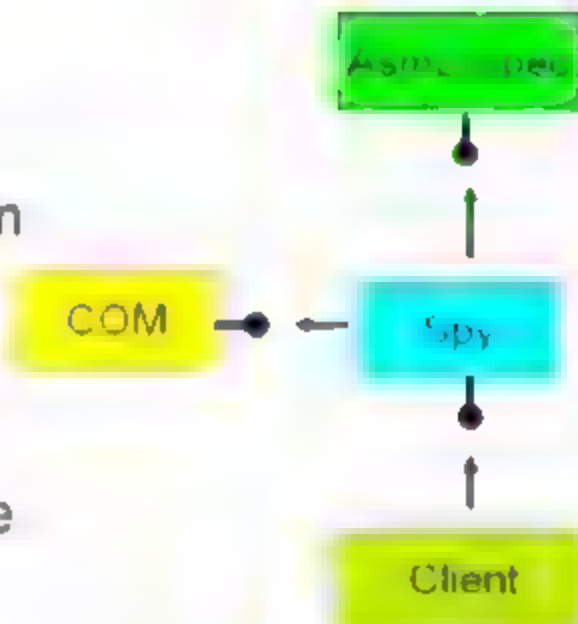
- Based on ASMs
- Readable executable testable
- Combines mathematical object-oriented component-oriented approaches
- Integrated into VS<sub>0</sub> (working on VS NET)
- Literate style Word and XML

# The Big Picture



## Previously...

- + Observational equivalence
- + No instrumentation required
- Non-determinism
- Callbacks
- State equivalence



## (Trivial) Example

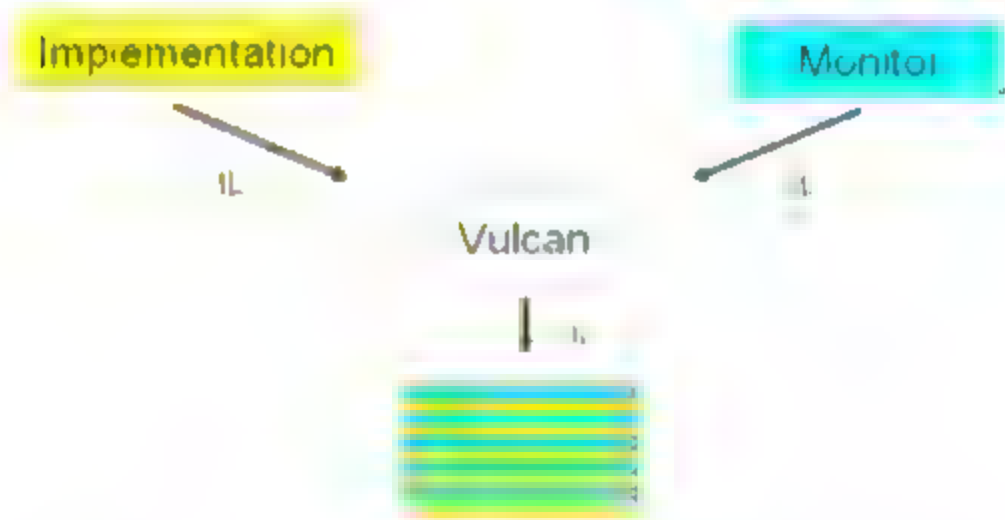
```
interface IMath {  
    double sqrt(double x);  
}
```

```
class CMath implements IMath {  
    double sqrt(double x)  
    { ... }  
}
```

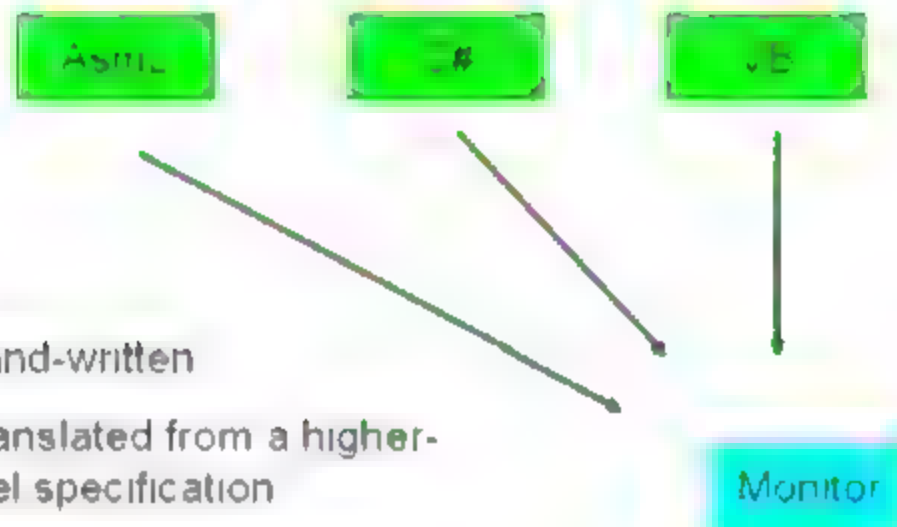
```
class IMathContract implements IMath {  
    bool sqrt_pre(double x)  
    { x >= 0; }  
    bool sqrt_post(double x)  
    { result < result + x }  
}
```



# Tighter Integration: NET



# Monitors



## (Trivial) Example

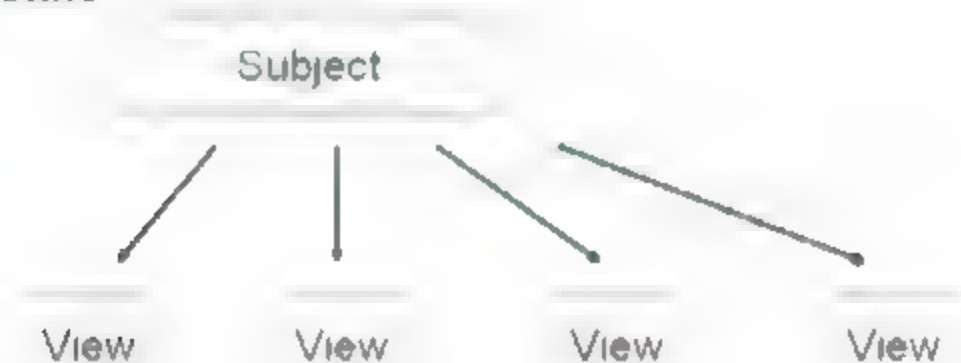
```
interface IMath {  
    double sqrt(double x);  
}
```

```
class CMath implements IMath {  
    double sqrt(double x)  
    { ... }  
}
```

```
class IMathContract {  
    CMath {  
        bool sqrt_pre(double x)  
        { x >= 0; }  
        bool sqrt_post(double x)  
        { result >= x; }  
    }  
}
```

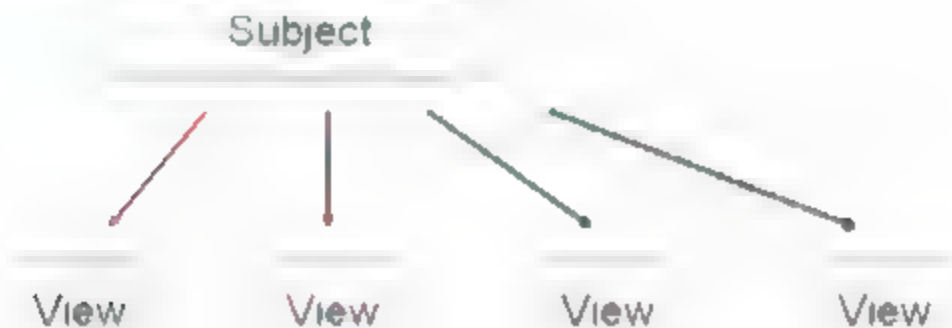
# Subject/View

**state**



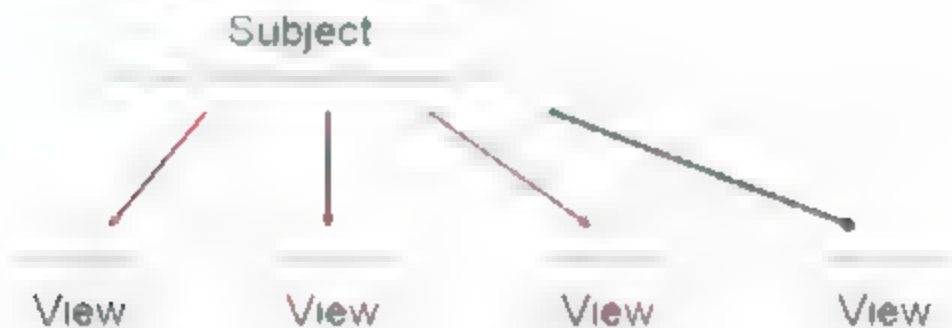
# Subject/View

state



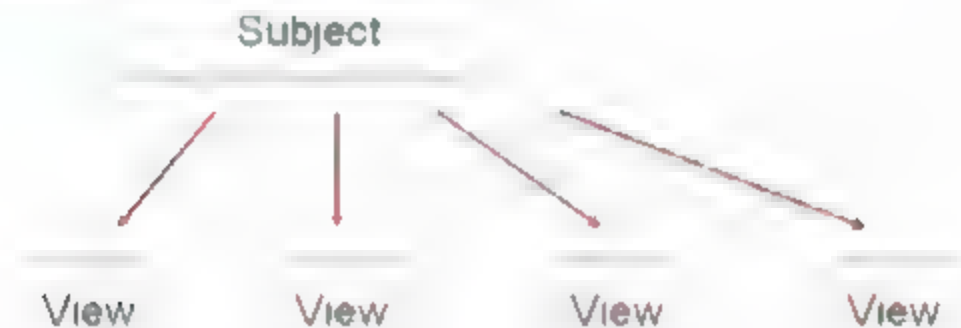
# Subject/View

state



# Subject/View

state



# Subject/View in AsmL

```
class Subject[T]  
  state as T  
  views as Set[View]  
  
  Get() as T = state  
  
  Set(x as T)  
    machine  
      state := x  
    step  
      forall v in views do  
        v.Update()
```




# Subject/View in AsmL

```
class Subject[T]  
  state as T  
  views as Set[View]  
  
  Get() as T = state  
  
  Set(x as T)  
  machine  
    state = x  
  step  
    forall v in views do  
      v.Update()
```

Notify **after** update



Enforce **forall**:  
all views see  
same state



# Subject/View in AsmL

```
class Subject[T]  
  state as T  
  views as Set[View]  
  
  Get() as T = state  
  
  Set(x as T)  
  machine  
    state = x  
  step  
    forall v in views do  
      v.Update()
```

Notify **after** update

Iterate **forall**:  
all views see  
same state

Mandatory call  
once per view

# Deriving the Monitor from AsmL

Set(x as T) =

machine state = x

step forall v in views do v.Update()

Set\_pre() { mcalls = { v.Update | v in views }, }

Set\_body() { state = x }

Set\_post() { assert(mcalls == {}) }

Update\_pre() { assert(state == x) }

Update\_post() { mcalls.remove(v.Update) }

# Summary: Runtime Verification

- Monitor design can be used for arbitrary purposes
- Monitor is generated automatically from AsmL
  - ┆ Can be added incrementally
  - ┆ Does not require model checking theorem proving etc

<http://asmf>

# The Big Picture

Programming

Specification and Verification

interface

extracts

specification

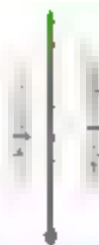
specification



implementation

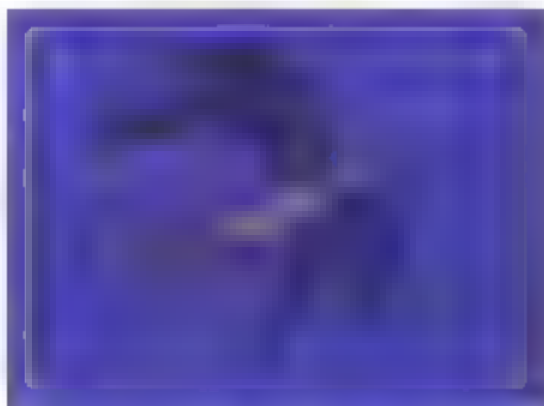
extracts

specification



model

# Using Garbage Collection to Improve Program Data Locality



Trishul Chilimbi

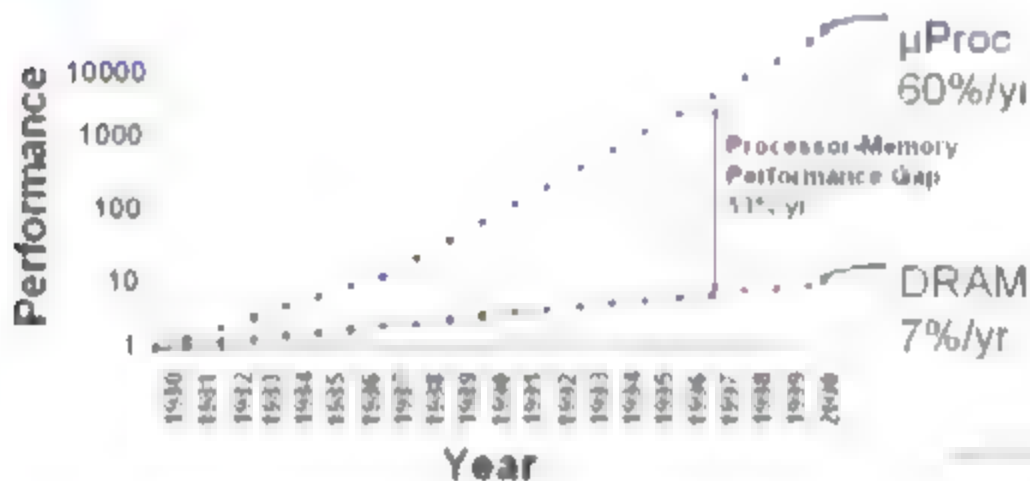
PPRC, Performance Monitoring & Analysis

MS Research

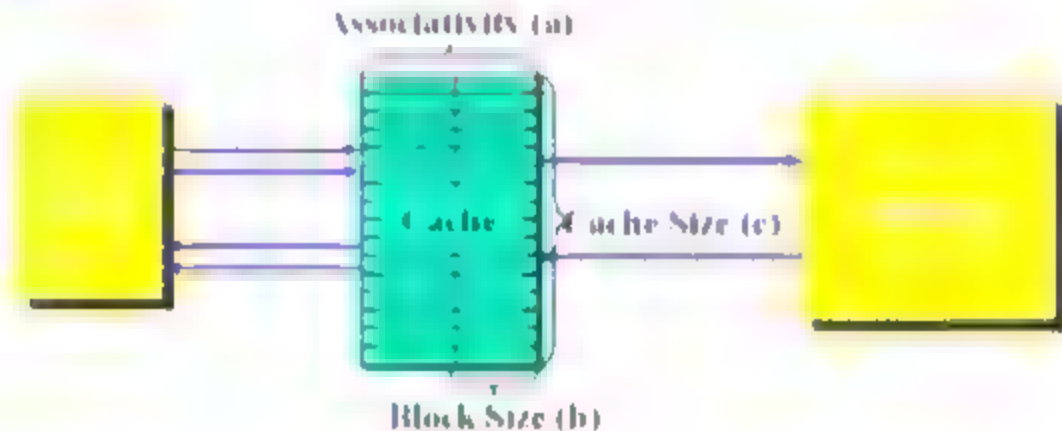
<http://research.microsoft.com/~trishulc/Daedalus.htm>



## Processor-Memory Imbalance



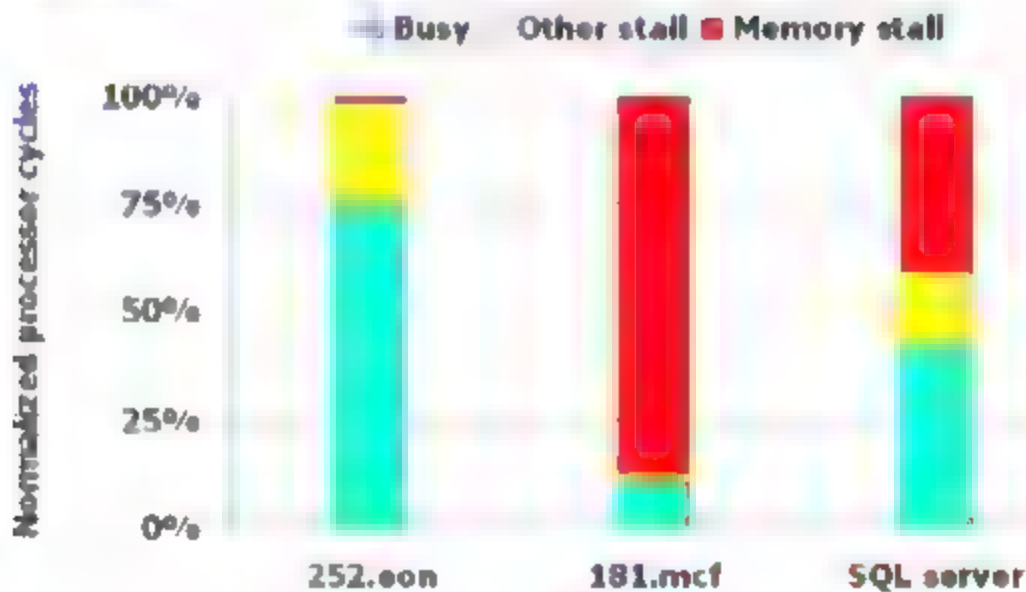
## Memory Cache: Reduce Latency







## Performance Impact





## Talk Outline

---

- 
- Motivation
- GC for implementing cache-conscious data layouts
- Low overhead data reference profiling
- Fast and accurate data layout determination



## Improving Memory System Performance

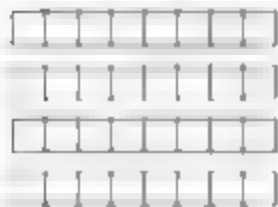
- **New Hardware**
  - Better caches
- **New Software**
  - Redesign code and structures
  - Cache-conscious algorithms
- **Our approach: Improve locality of existing software automatically**
  - Cache-conscious data layouts



## Cache-Conscious Data Layout



Random Layout



Cache  
Conscious



## Cache-Conscious Data Layout



### Random Layout

Cache  
Consistent



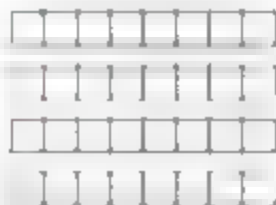
## Cache-Conscious Data Layout



Random Layout



Cache  
Conscious



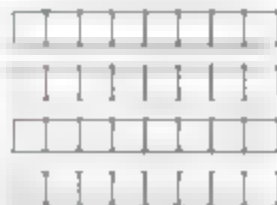
# Cache-Conscious Data Layout



Random Layout



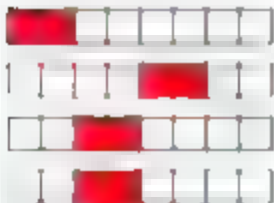
Cache  
Conscious



# Cache-Conscious Data Layout



Random Layout



Cache  
Conscious



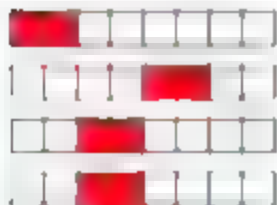




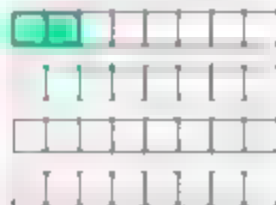
## Cache-Conscious Data Layout



Random Layout



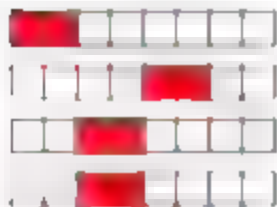
Cache-Conscious



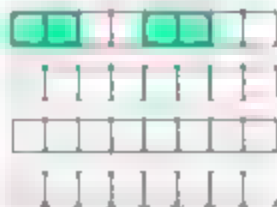
# Cache-Conscious Data Layout

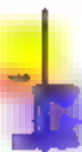


Random Layout



Cache Conscious





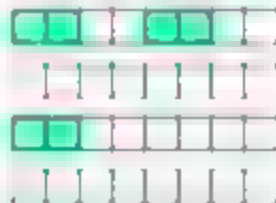
## Cache-Conscious Data Layout



Random Layout



Cache  
Conscious

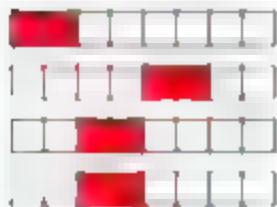




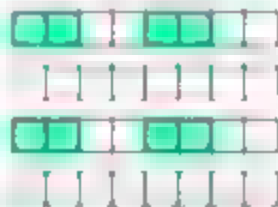
## Cache-Conscious Data Layout



Random Layout



Cache-Conscious





## Talk Outline

- 
- GC for implementing cache-conscious data layouts
  - Structure reorganization
  - Hot/cold splitting
  - Limitations
- Low-overhead data reference profiling
- Fast and accurate data layout determination



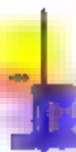
## Copying Garbage Collection

Memory Pool

Free Space

1. Marking Phase





# Copying Garbage Collection

Memory

Space





# Copying Garbage Collection

1. Marking

2. Copying

3. Sweeping







# Copying Garbage Collection





# Copying Garbage Collection

Memory layout

Memory layout

Memory layout





## Copying Garbage Collection

Memory

Free

Free Space





# Copying Garbage Collection

Before

After

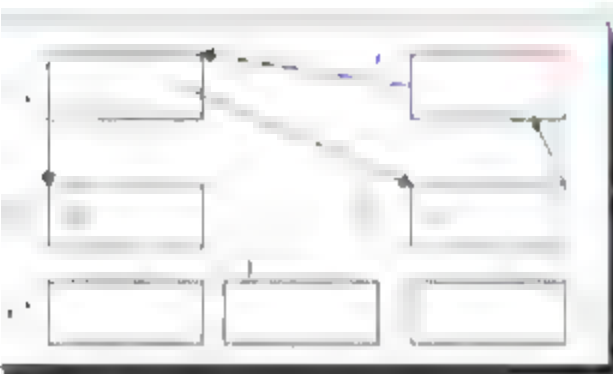


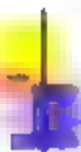


# Copying Garbage Collection

Initial State

Final State

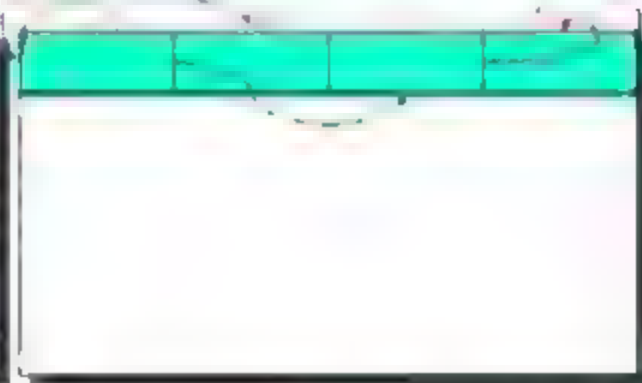




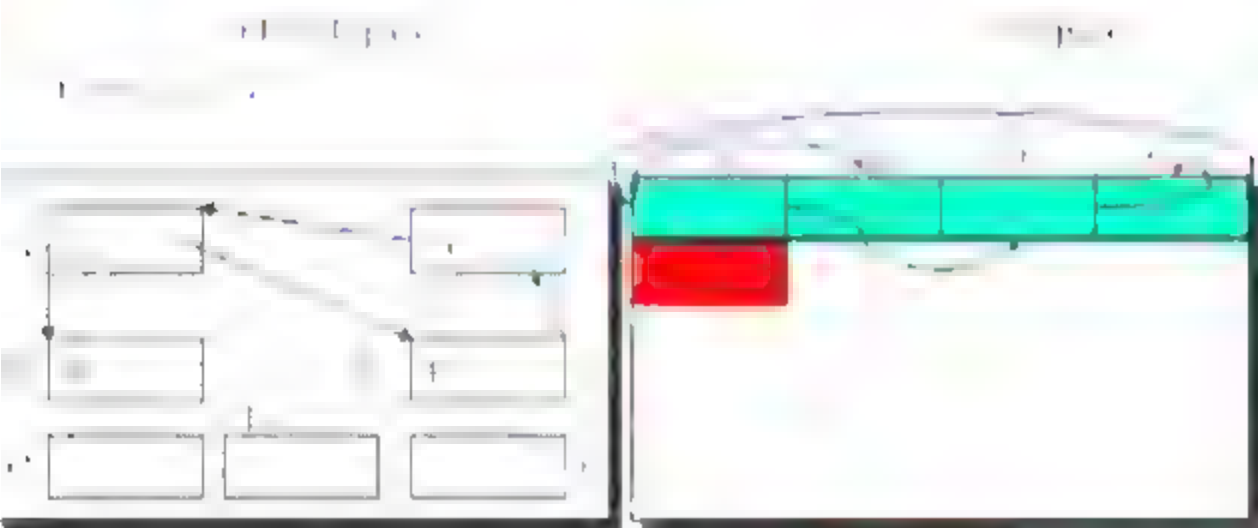
## Copying Garbage Collection

Initial Space

Copy Space



# Copying Garbage Collection

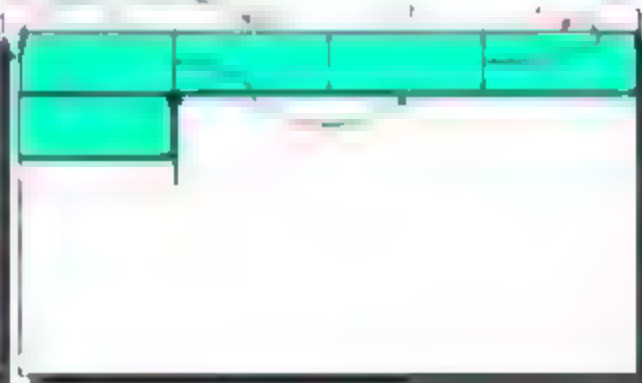
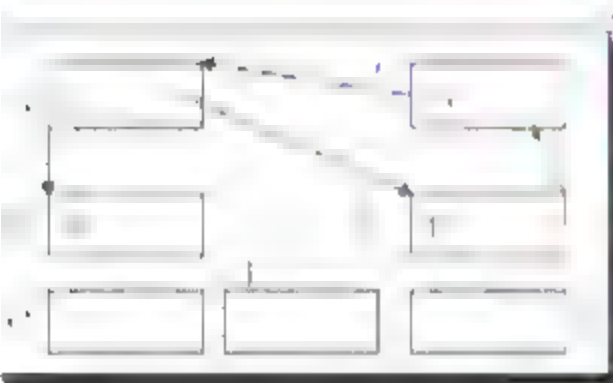




## Copying Garbage Collection

Original graph

Copy





# Copying Garbage Collection



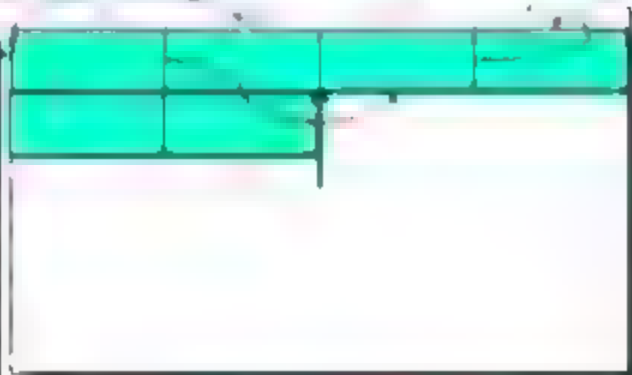


## Copying Garbage Collection

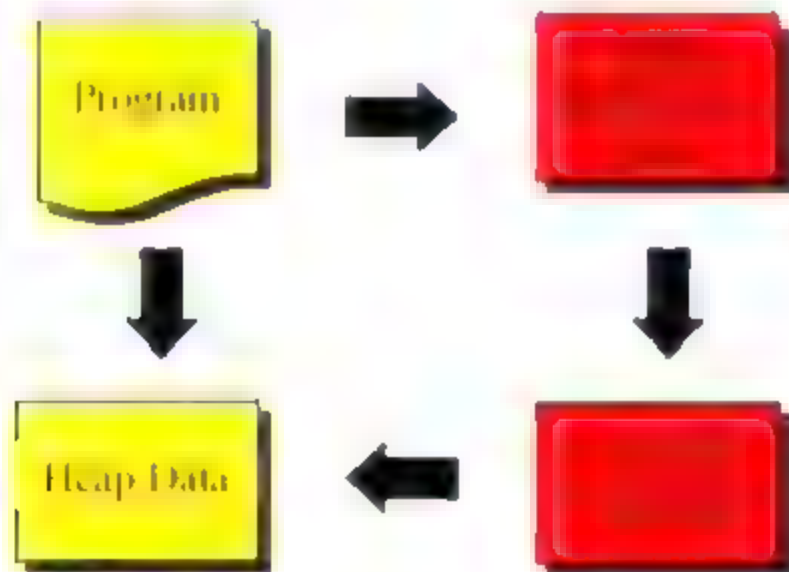
Initial State

Final State

1. Marking



# Using Copying GC for Cache Conscious Reorganization

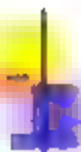




## Object Access Buffer



Low  
cost  
used



## Object Access Buffer

Front



Back  
view

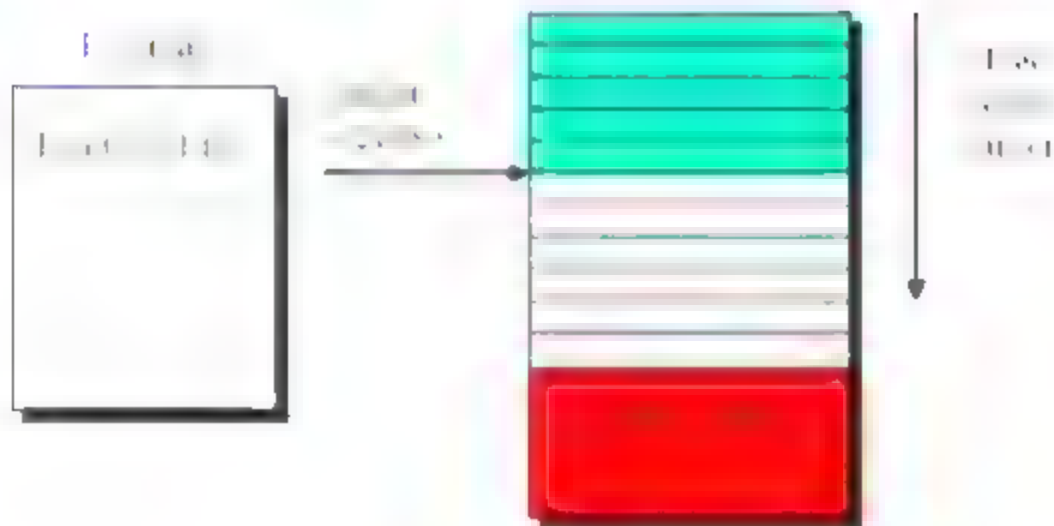


## Object Access Buffer

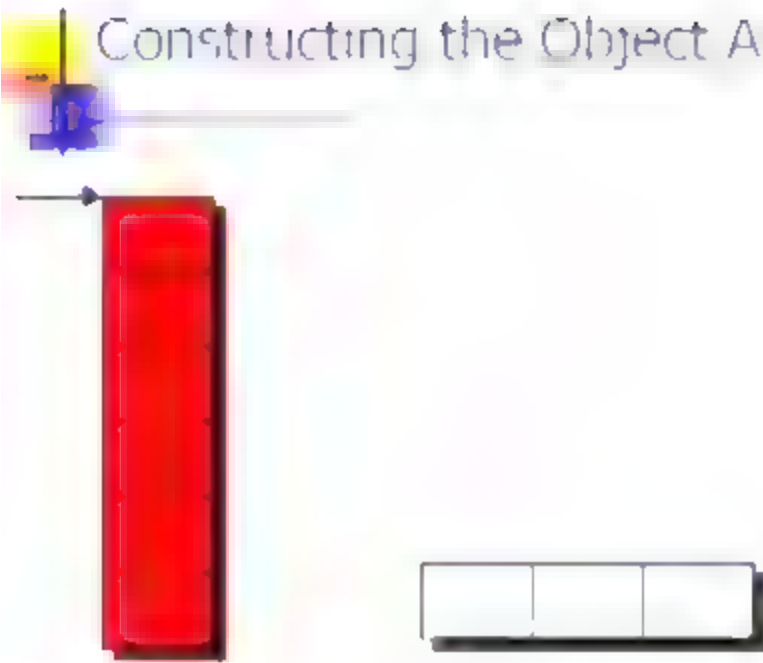




## Object Access Buffer



## Constructing the Object Affinity Graph

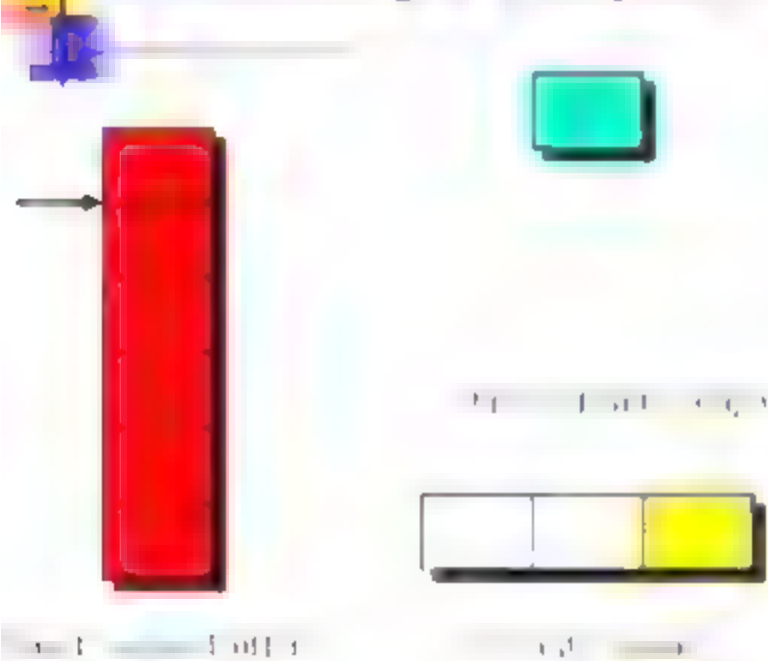


Object Access Buffer

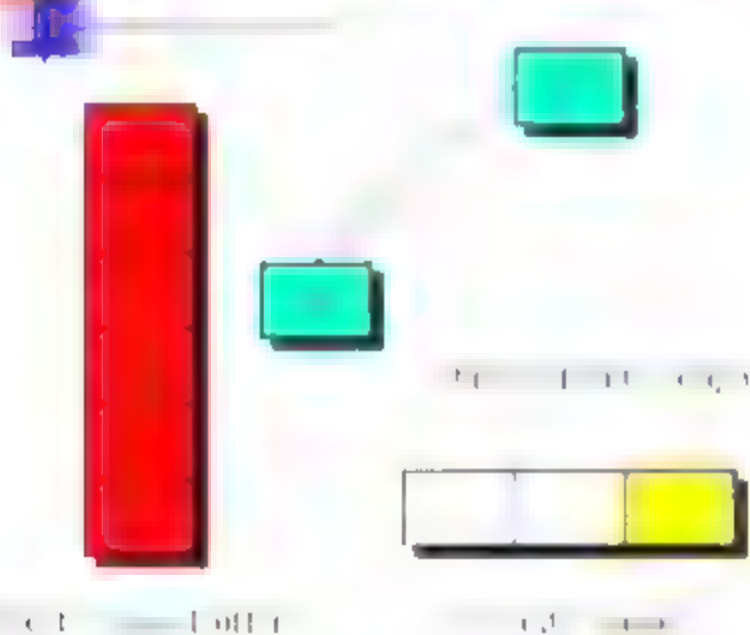
L-wait



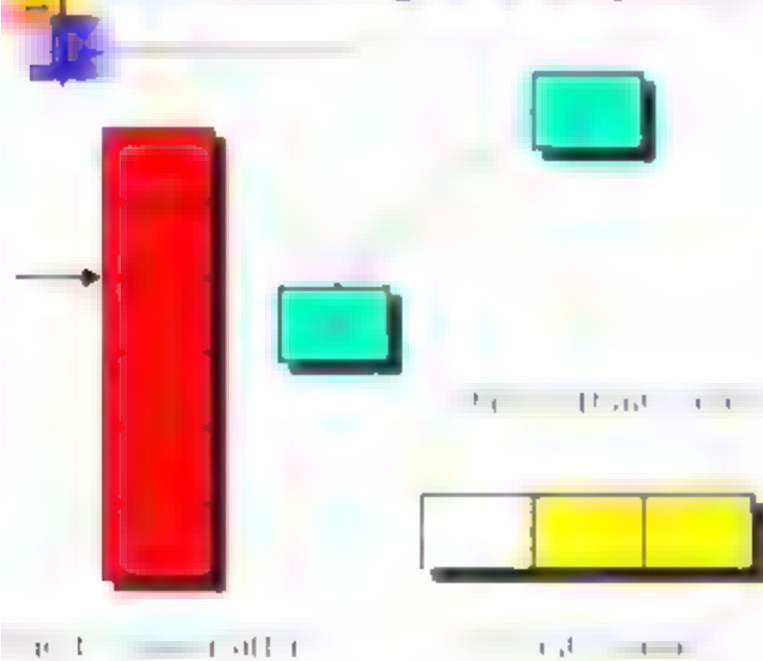
## Constructing the Object Affinity Graph



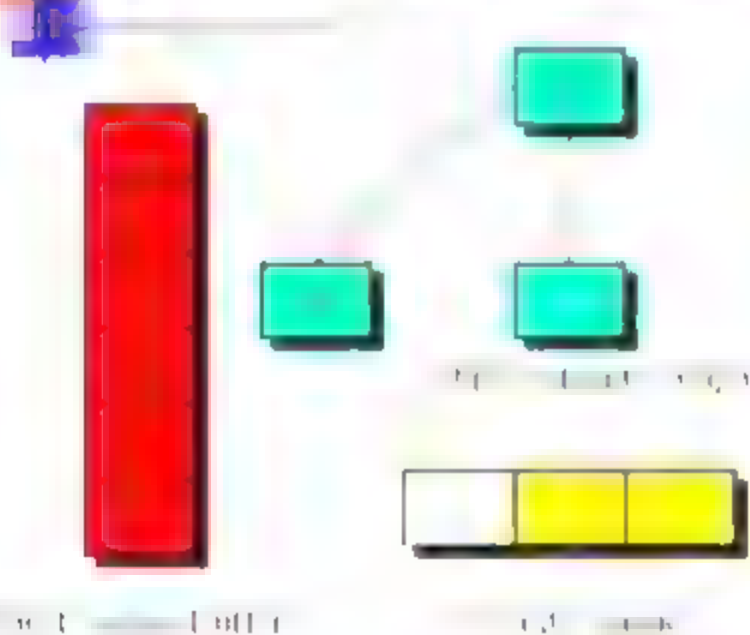
# Constructing the Object Affinity Graph



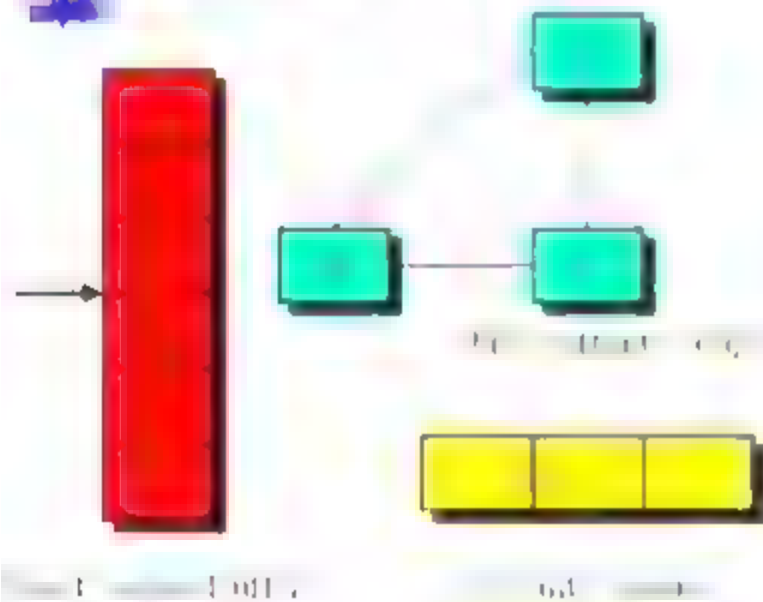
# Constructing the Object Affinity Graph



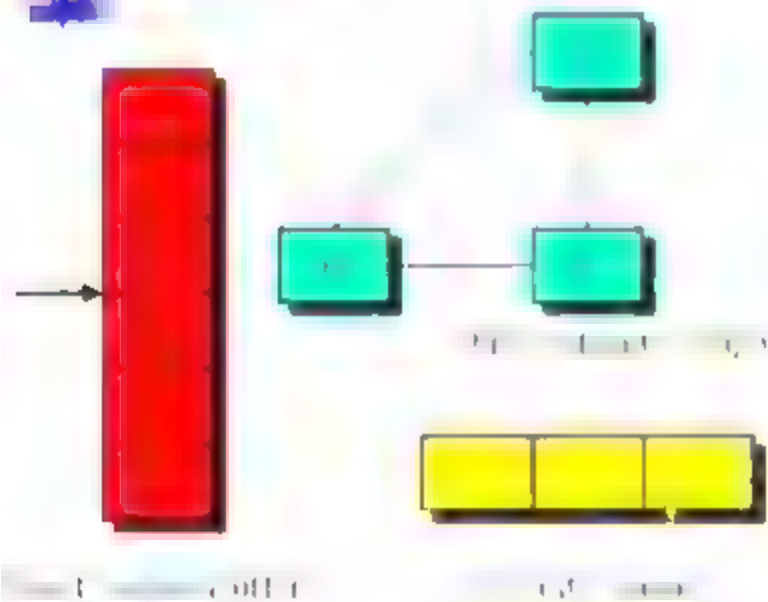
# Constructing the Object Affinity Graph



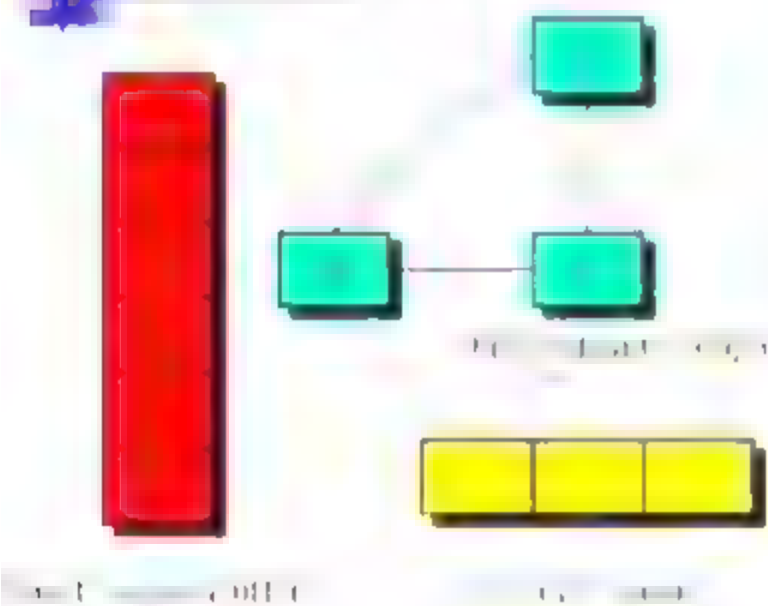
# Constructing the Object Affinity Graph



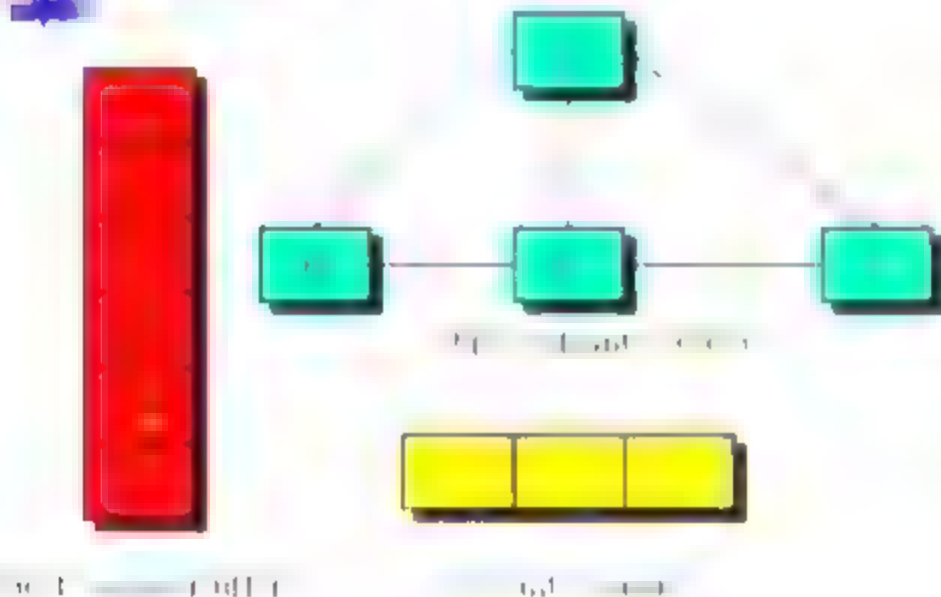
## Constructing the Object Affinity Graph



## Constructing the Object Affinity Graph

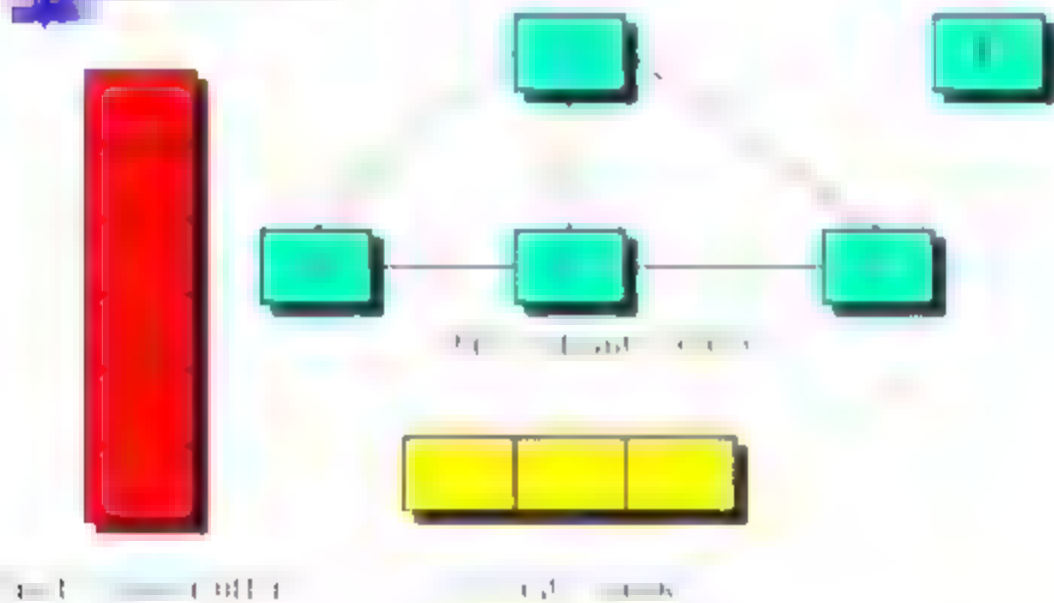


# Constructing the Object Affinity Graph





# Constructing the Object Affinity Graph



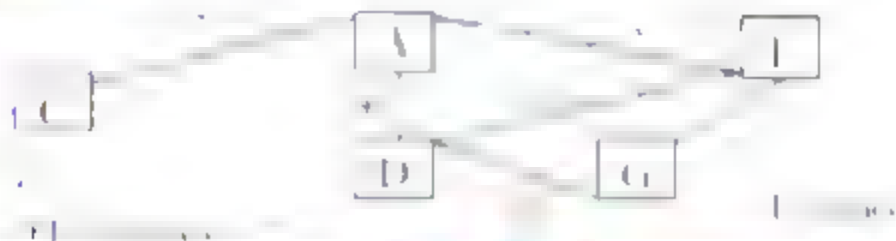
# Constructing the Object Affinity Graph



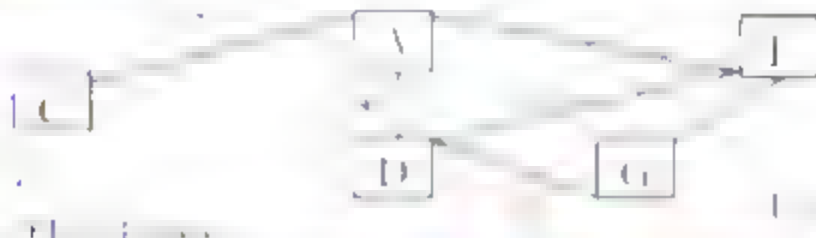
## Cache-Conscious Copying



## Cache-Conscious Copying



# Cache-Conscious Copying



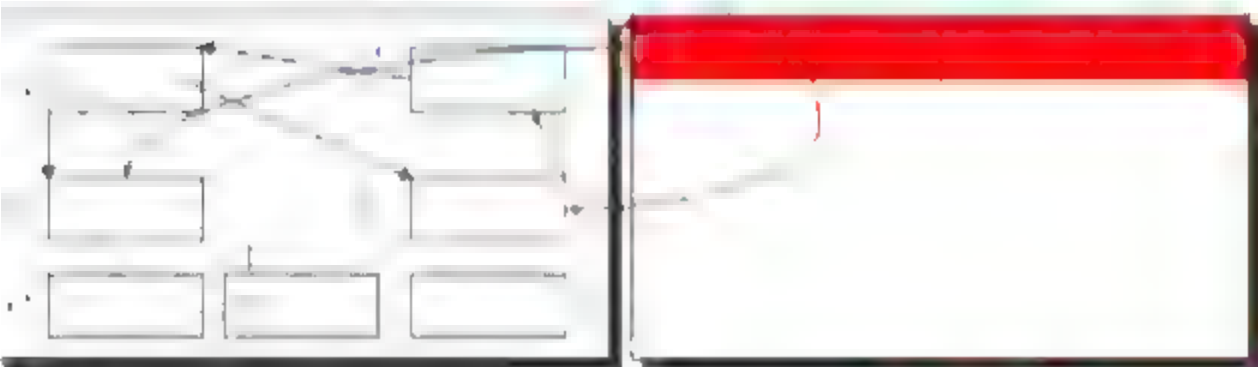
## Cache-Conscious Copying



## Cache-Conscious Copying

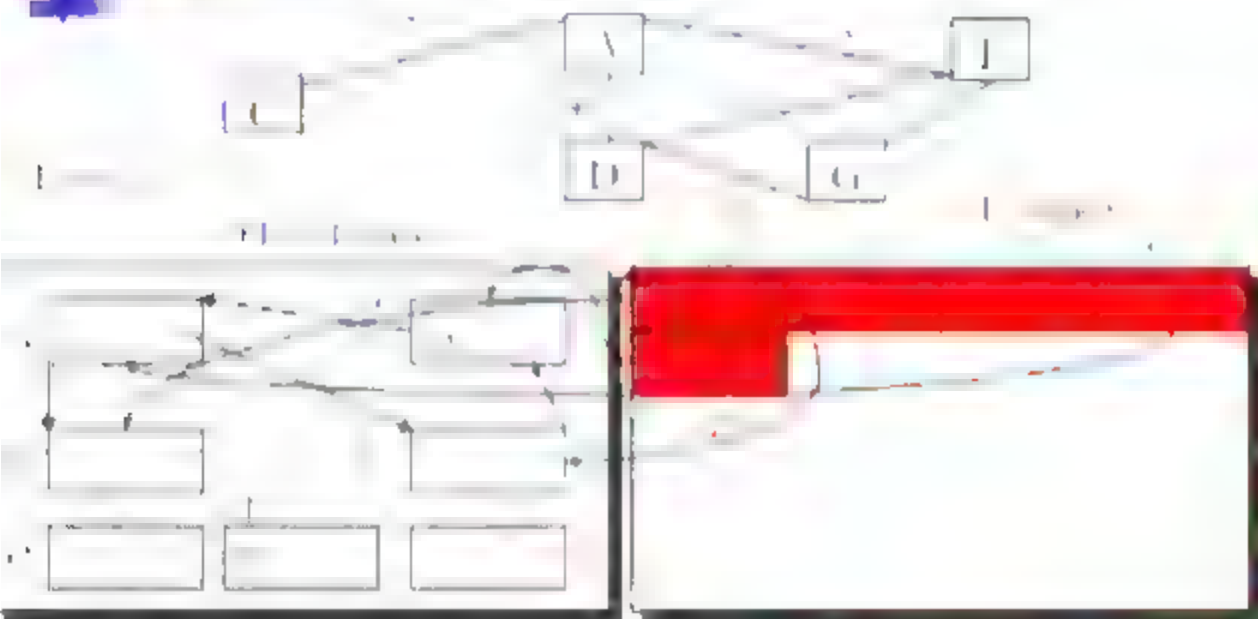


## Cache-Conscious Copying





## Cache-Conscious Copying



# Cache-Conscious Copying



# Cache-Conscious Copying





# Cache-Conscious Copying

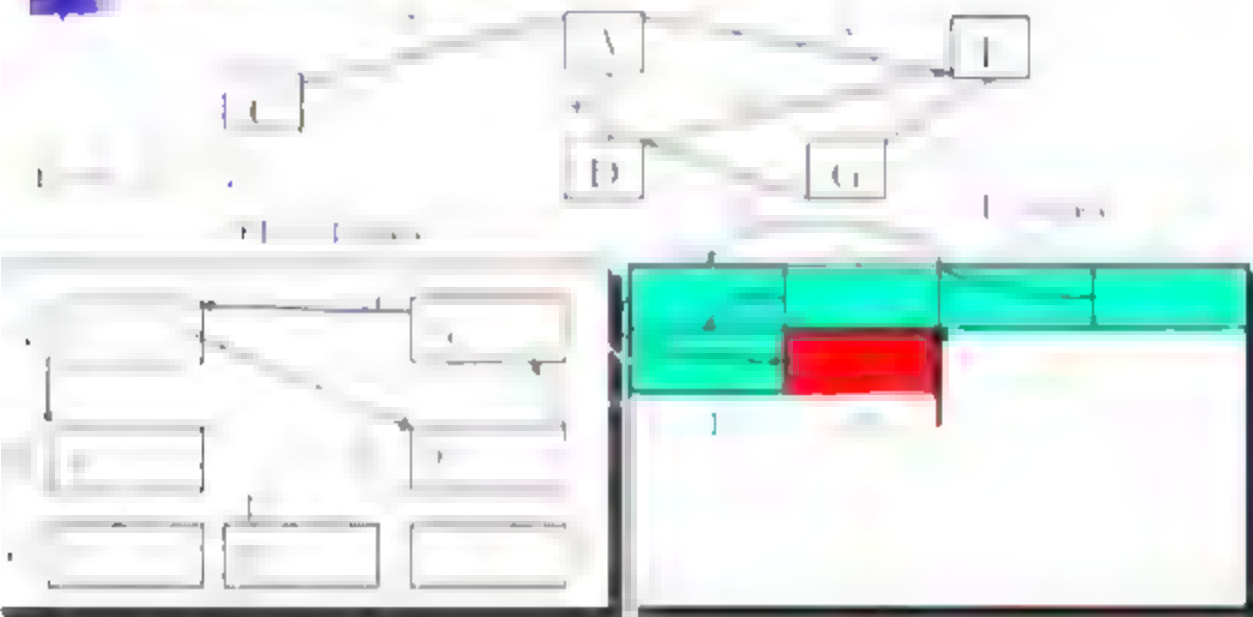


# Cache-Conscious Copying



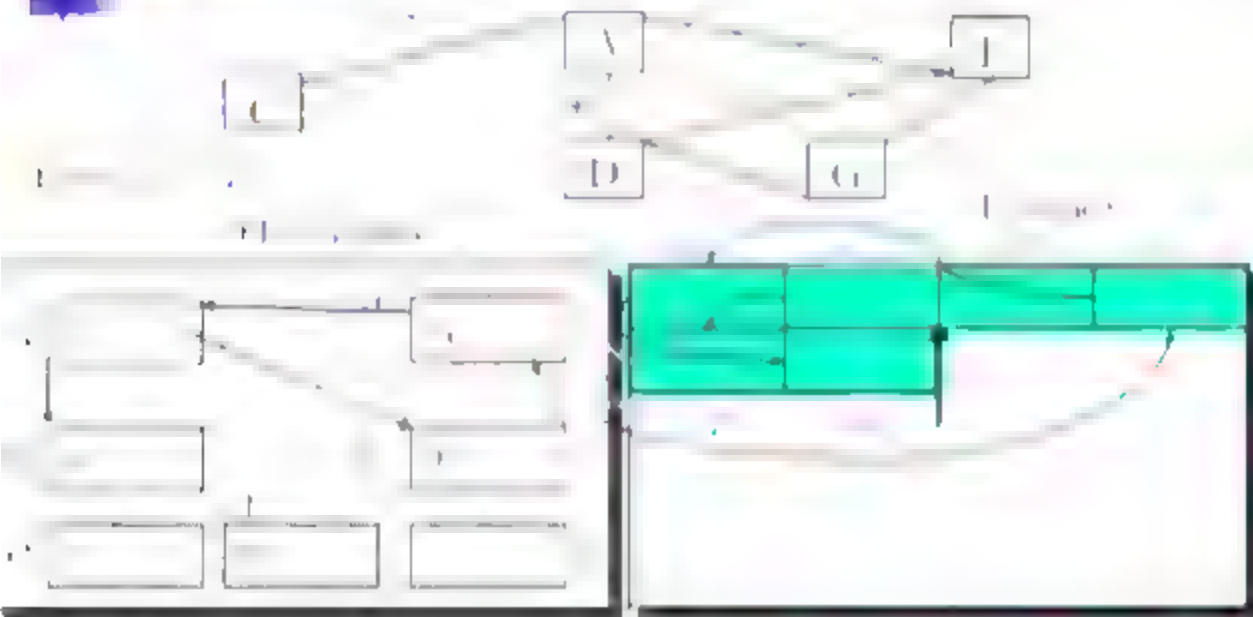


## Cache-Conscious Copying





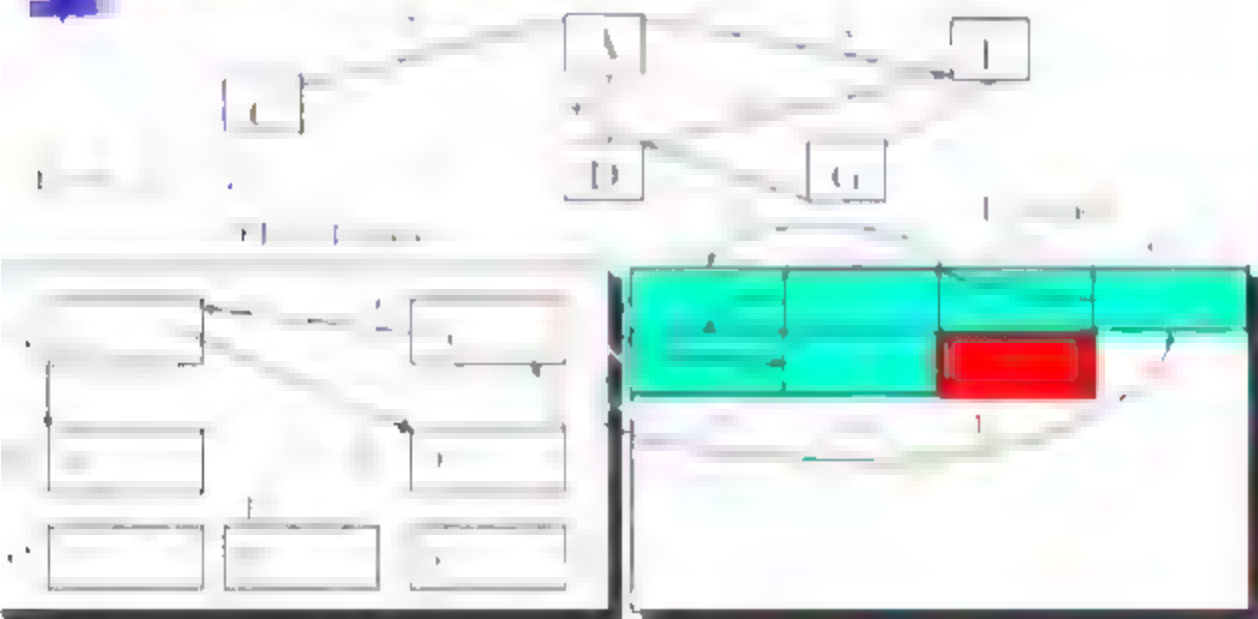
# Cache-Conscious Copying



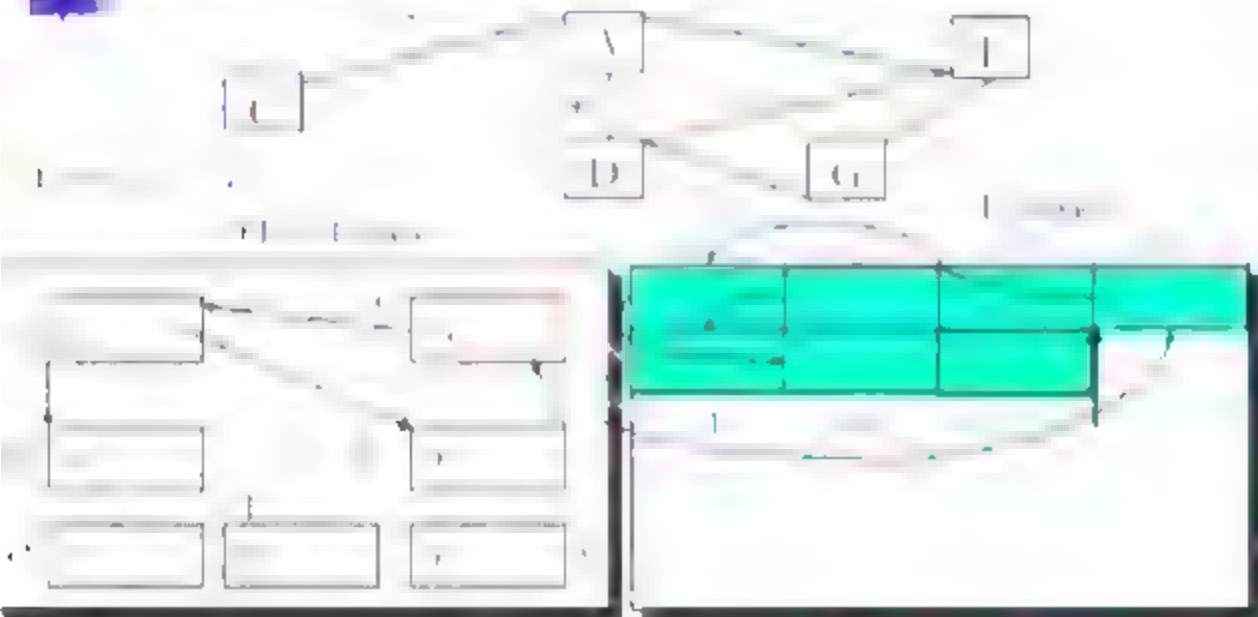
# Cache-Conscious Copying



# Cache-Conscious Copying

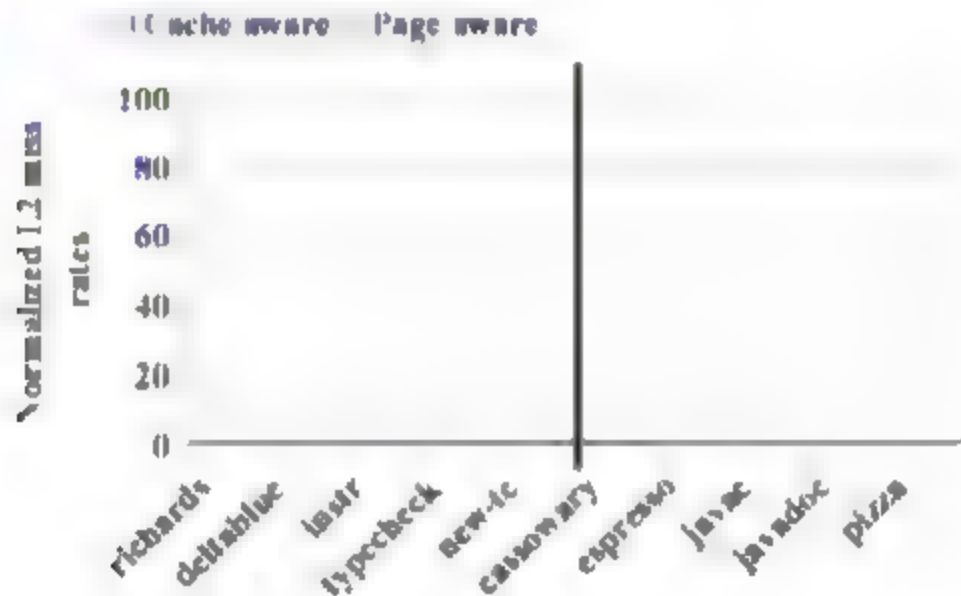


## Cache-Conscious Copying



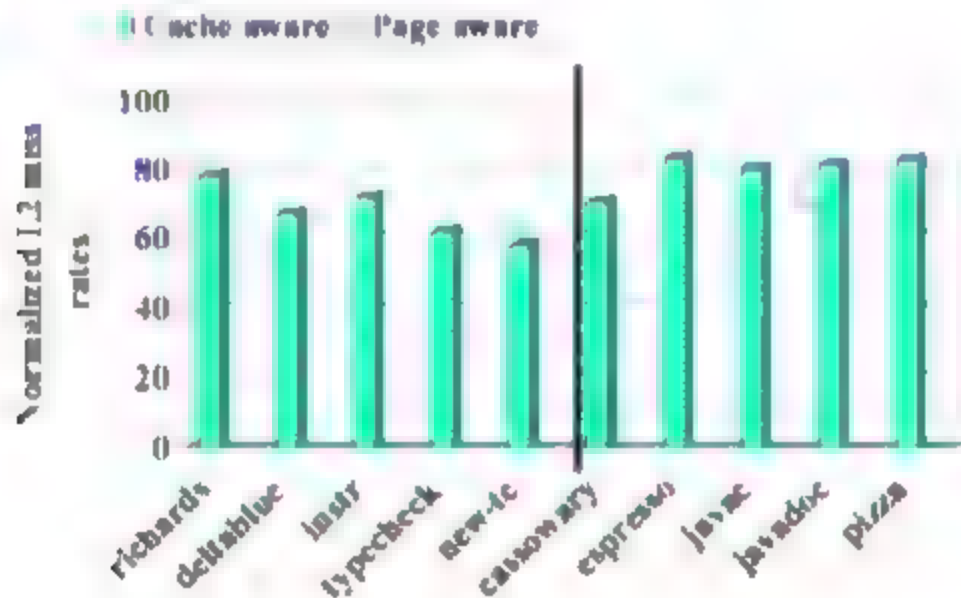


# Cache-Conscious Reorganization

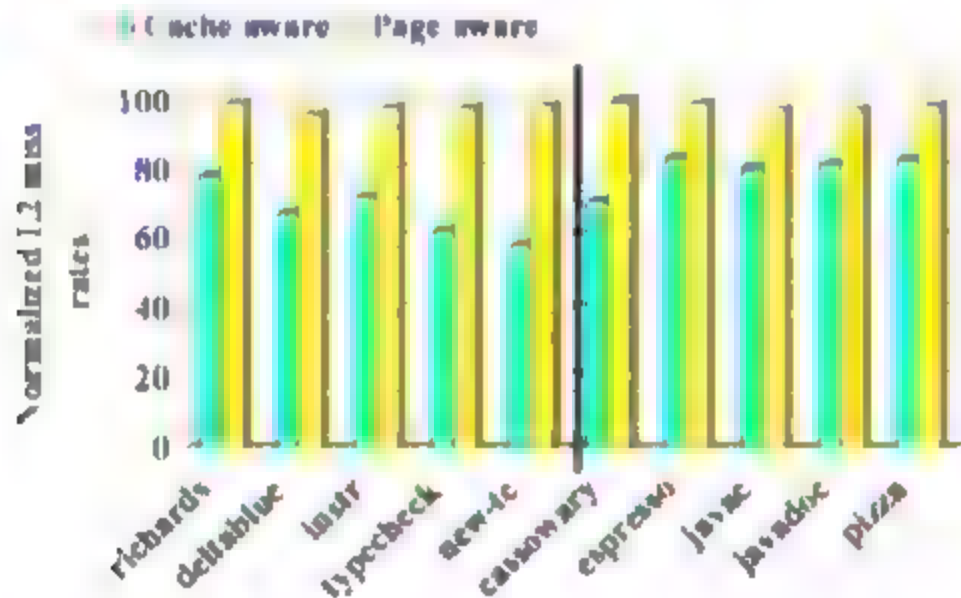




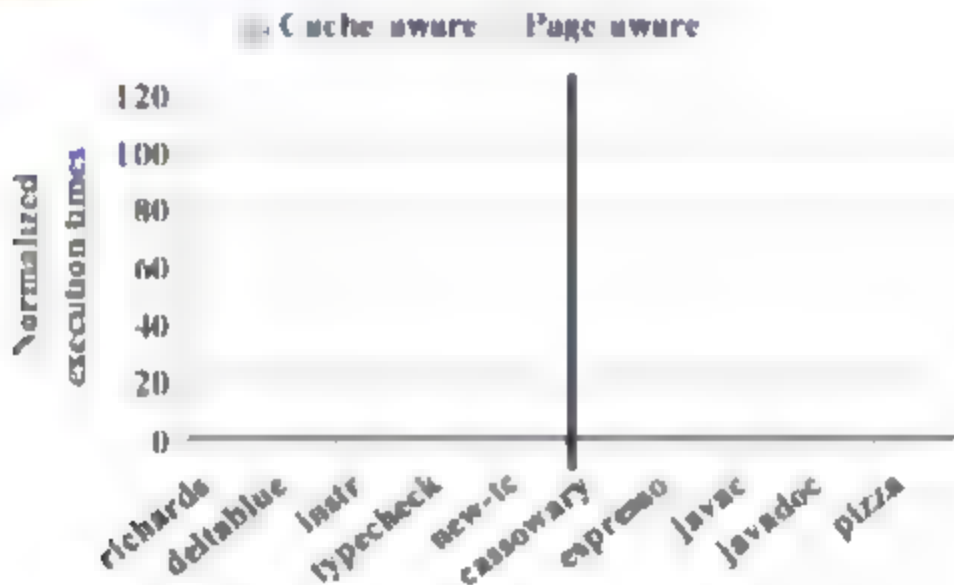
# Cache-Conscious Reorganization



# Cache-Conscious Reorganization

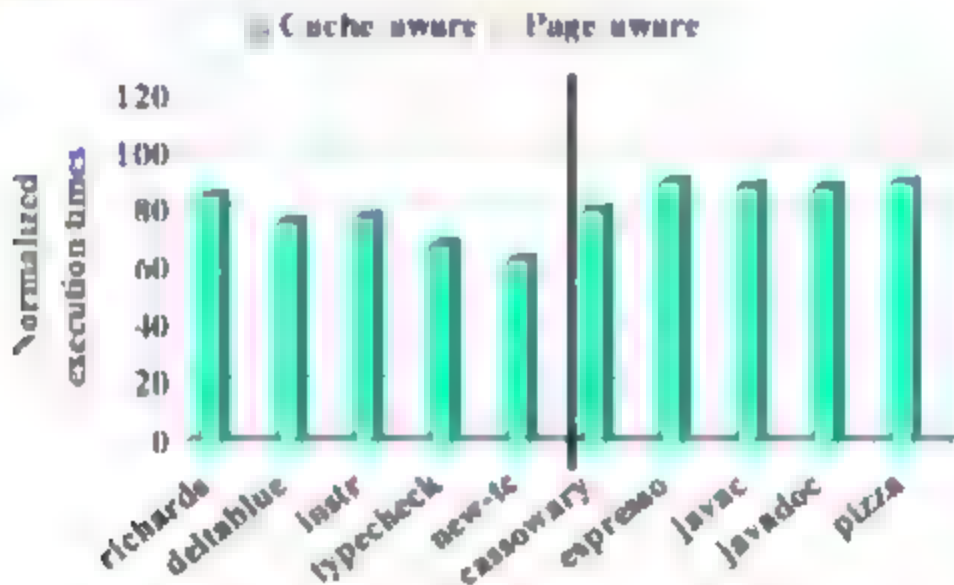


# Cache-Conscious Reorganization



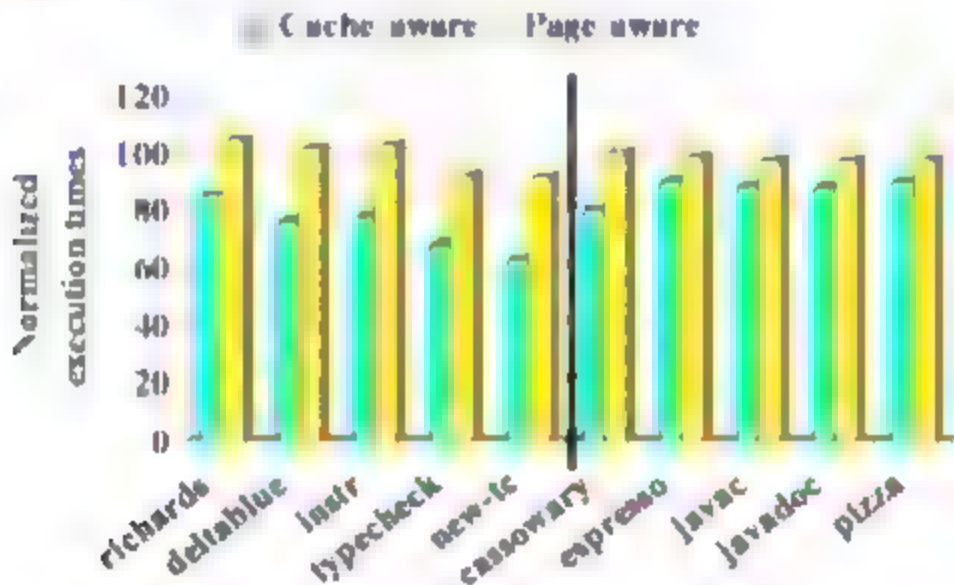


# Cache-Conscious Reorganization





## Cache-Conscious Reorganization



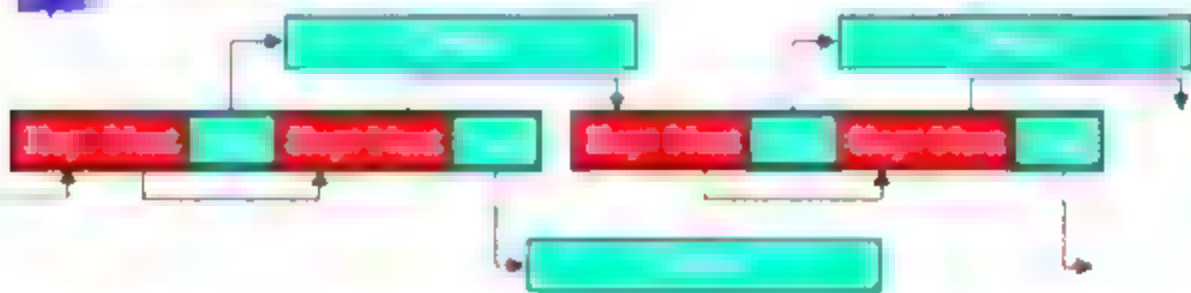
## Hot/Cold Structure Splitting

Cache block 1

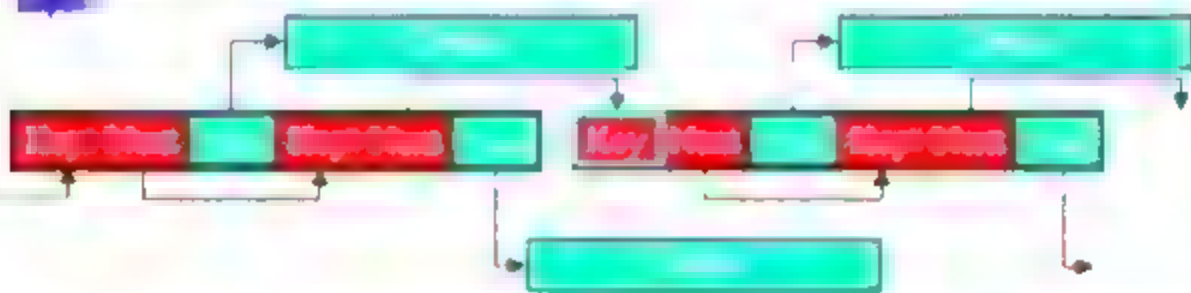
Cache block 2



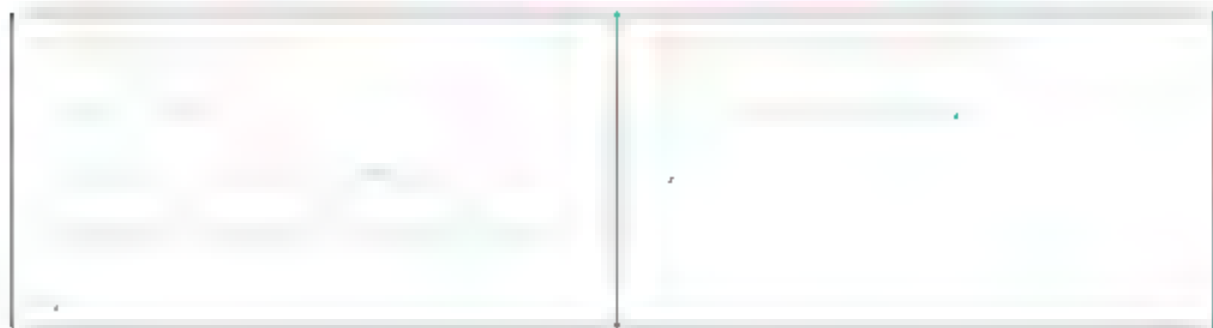
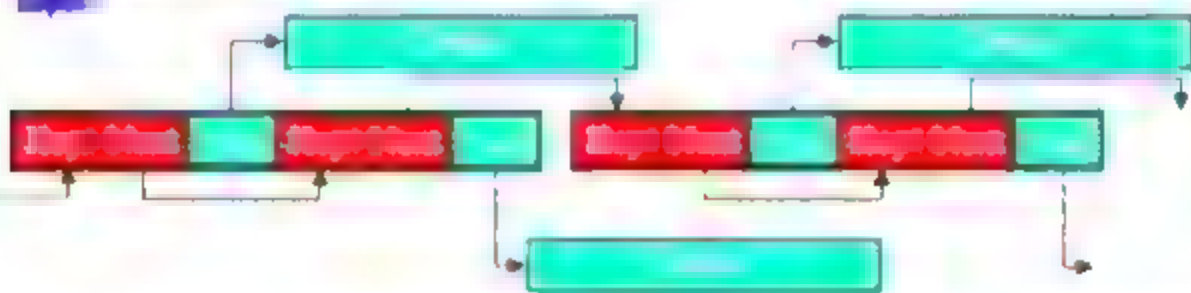
## Hot/Cold Structure Splitting



## Hot/Cold Structure Splitting



## Hot/Cold Structure Splitting

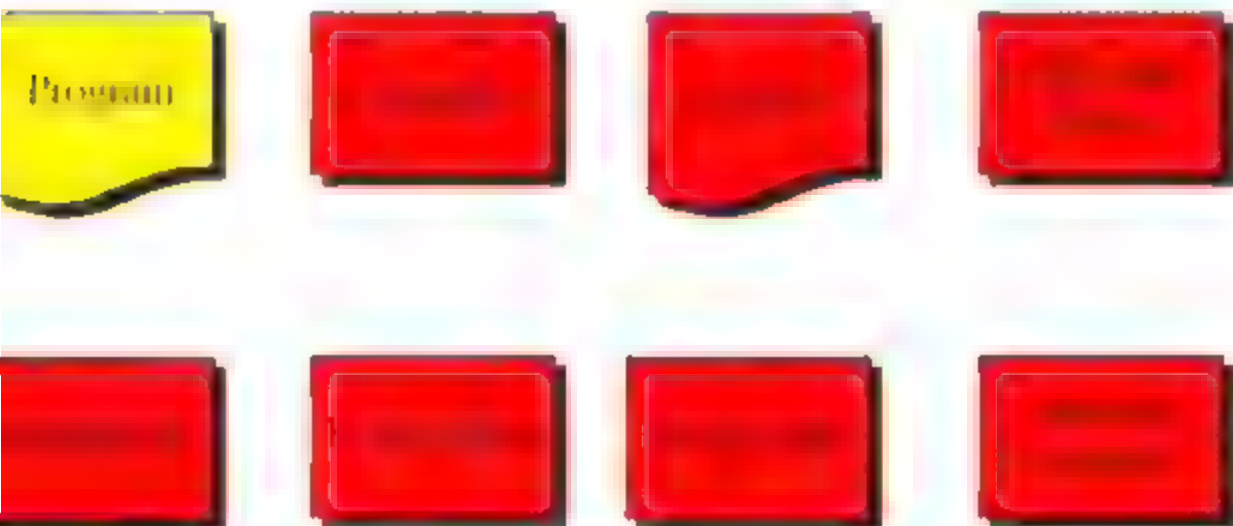




## Hot/Cold Structure Splitting Algorithm

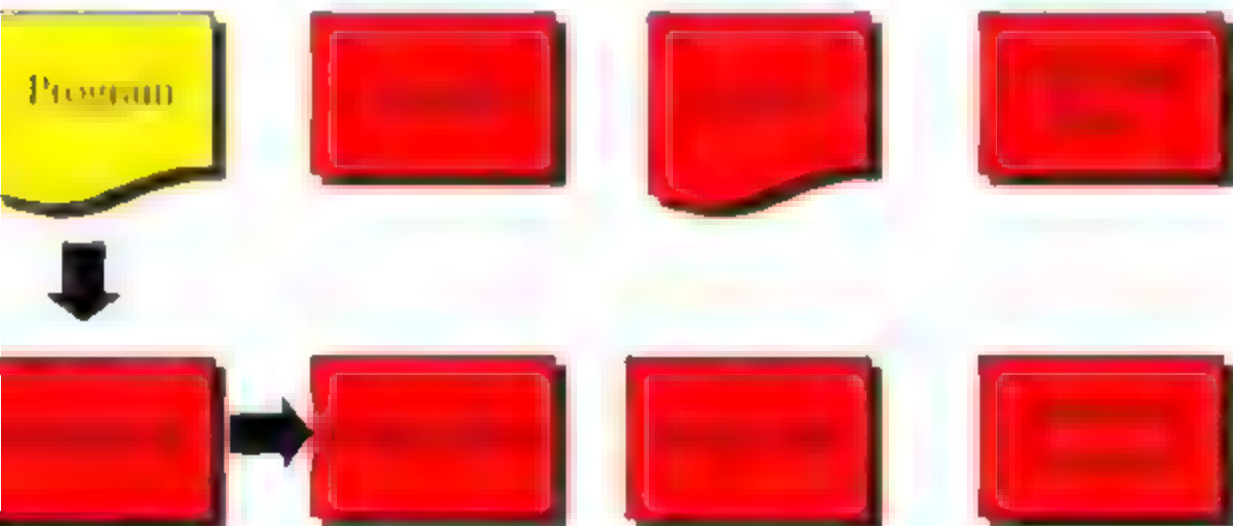
- Profile field access counts
- Identify hot structure fields
- Remove cold fields & place in new structure (link from old)
- Insert additional allocation call for new structure
- Access cold fields through new structure

# Cache-Conscious Reorganization (with Structure Splitting)

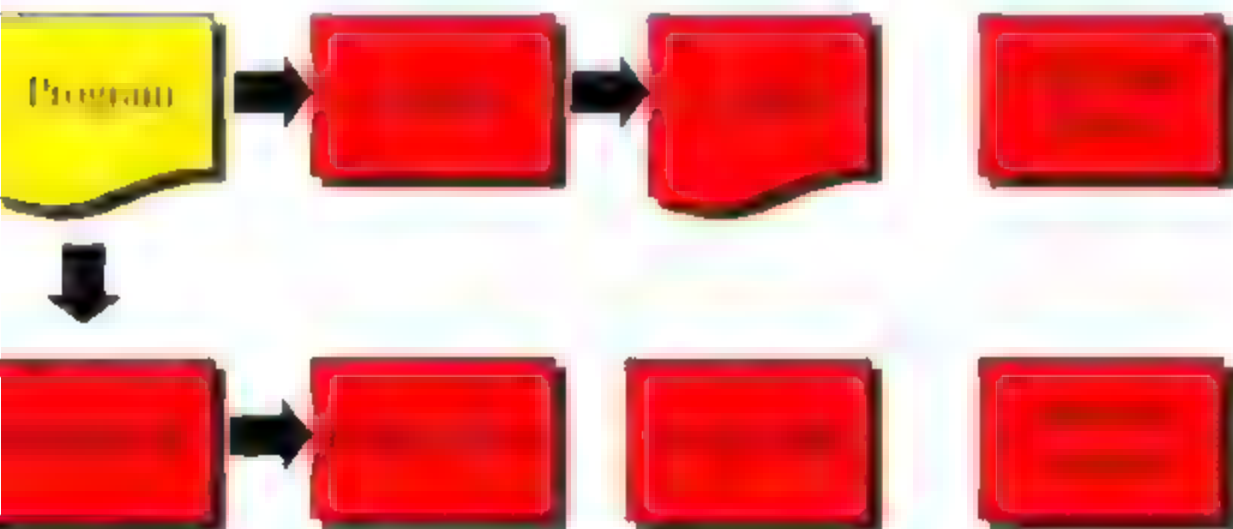




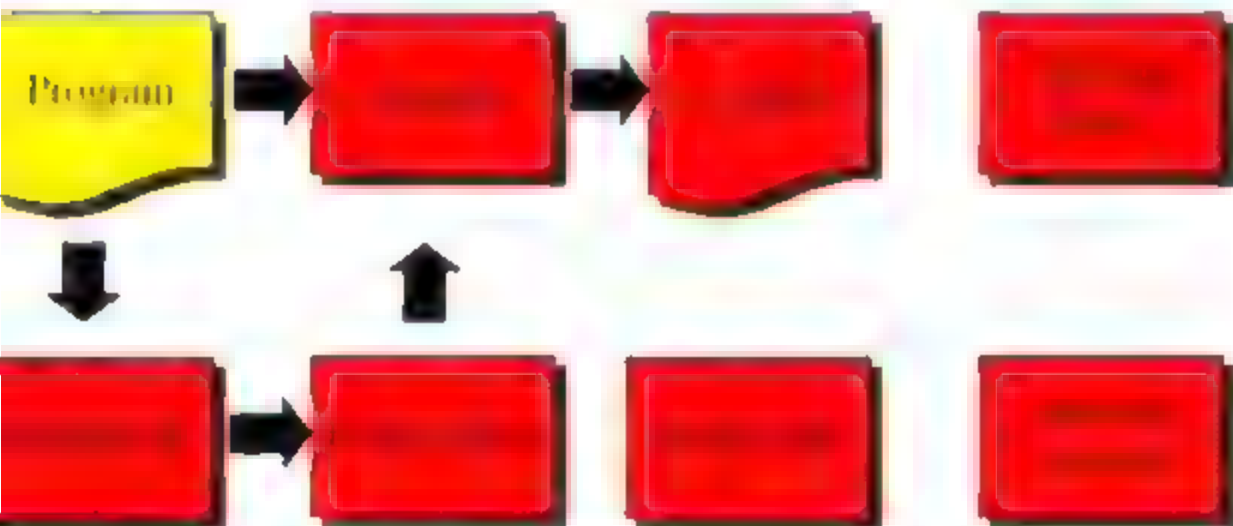
## Cache-Conscious Reorganization (with Structure Splitting)



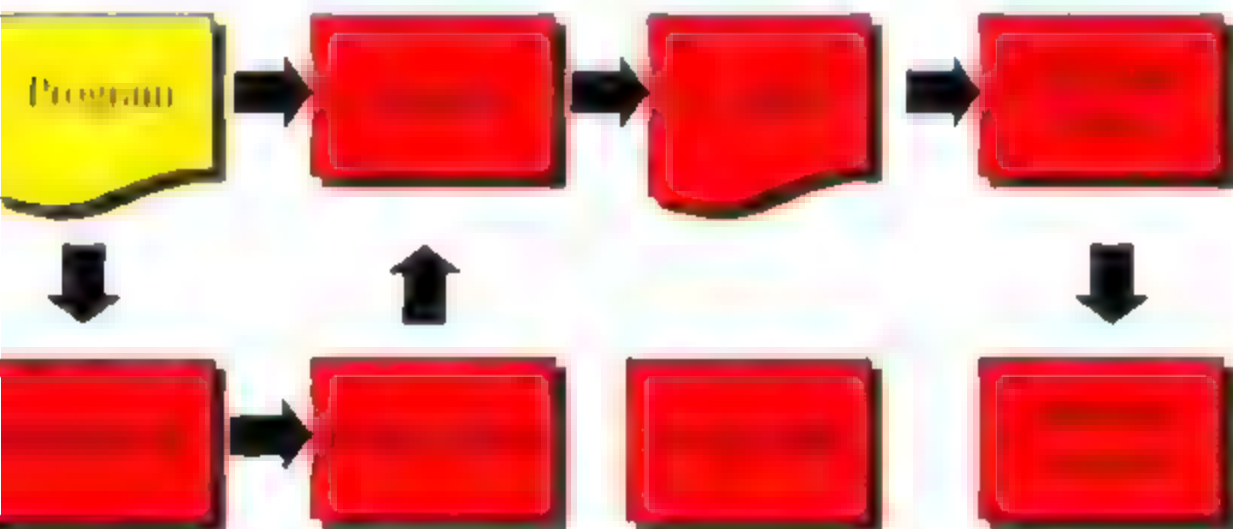
## Cache-Conscious Reorganization (with Structure Splitting)



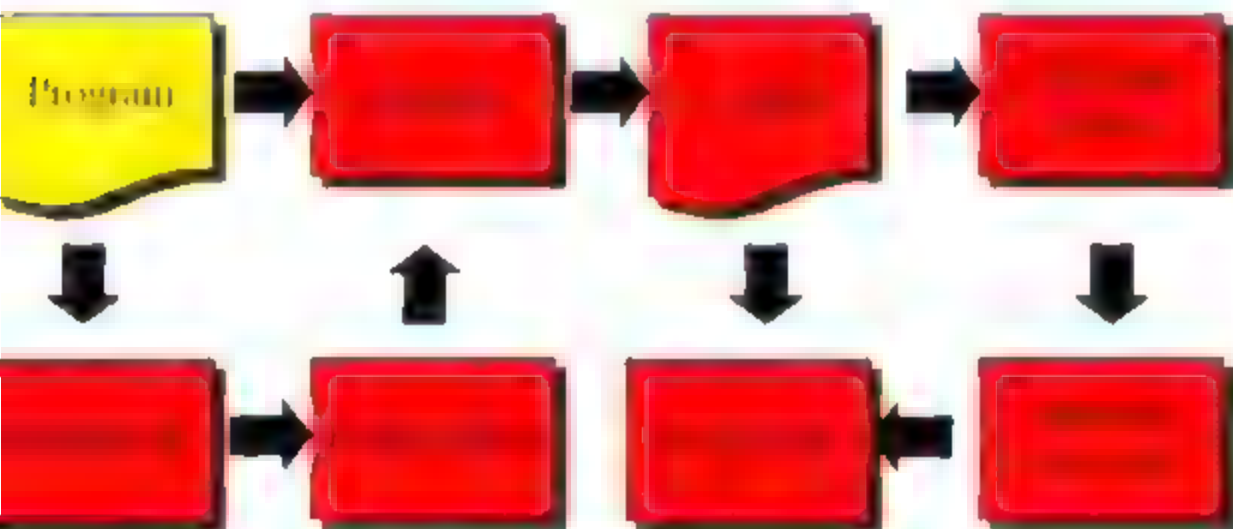
## Cache-Conscious Reorganization (with Structure Splitting)



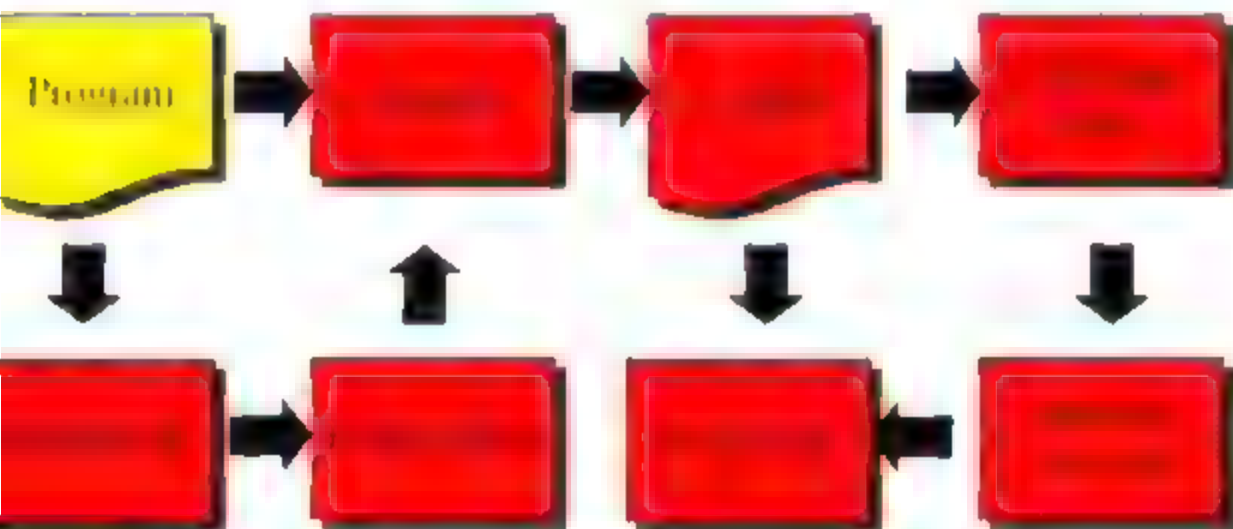
# Cache-Conscious Reorganization (with Structure Splitting)



# Cache-Conscious Reorganization (with Structure Splitting)



## Cache-Conscious Reorganization (with Structure Splitting)

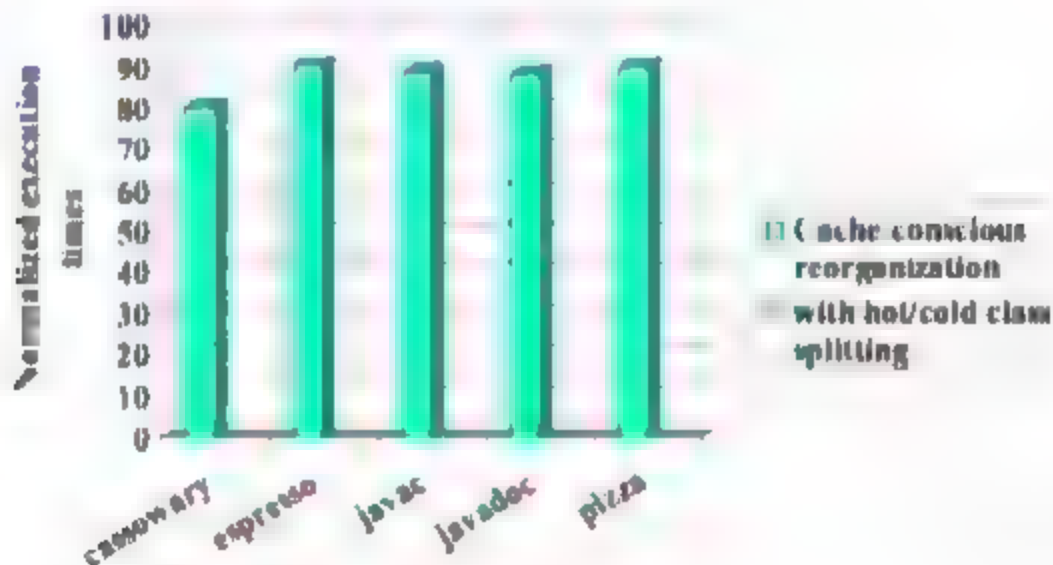


## Hot/Cold Class Splitting



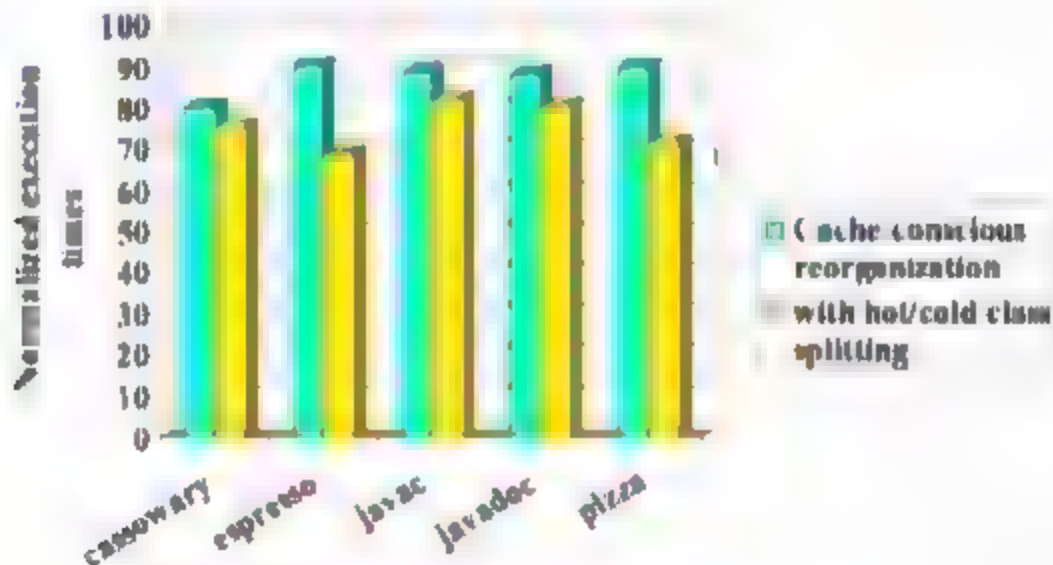


## Hot/Cold Class Splitting





## Hot/Cold Class Splitting





## Limitations

---

- Low-overhead data reference profiling
  - Requires reserving register
  - Potentially inaccurate compiler analysis
- Object Affinity Graph
  - Approximates temporal relationships
  - Higher accuracy increases processing cost

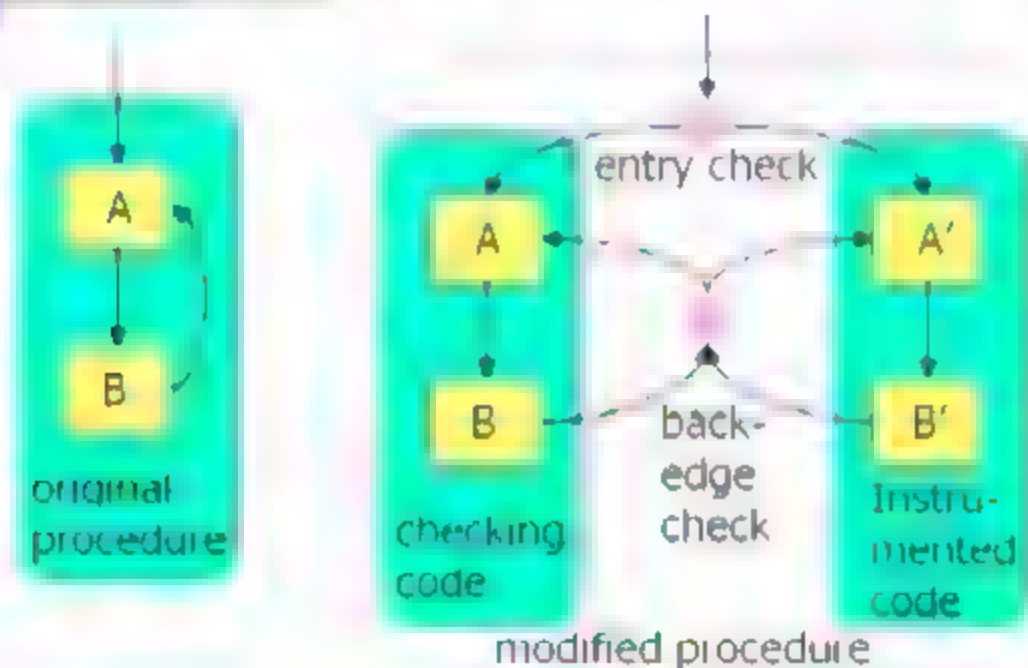


## Talk Outline

---

- 
- 
- Low-overhead data reference profiling
- Fast and accurate data layout determination

## Low-overhead data reference profiling





## Low-overhead data reference profiling

- 0.1% sampling rate provides high accuracy
- 5% overhead at this sample rate
- Base case is highly optimized x86 code



## Talk Outline

---

- 
- 
- 
- Fast and accurate data layout determination

## Exploitable Locality

- Hot Data Streams:  
Data object sequences that frequently repeat
- Hot=>reference skew + Repetition=>regularity
- Analogous to hot program paths

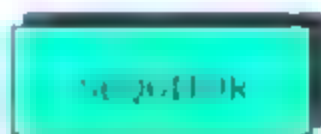
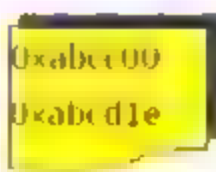
AB, A, AC, EF, AB, AD, CE, EF, AB, A, AC, EF, DD, CE, EF, B, AB, A

Hot Data Streams: **AB** **EF**



## Computing Exploitable Locality

Data  
Reference  
Trace



Regularity detector

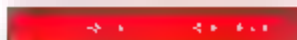


Hot Data Streams

a




b



Whole Program Streams  
(WPS)






## Fast, accurate data layout determination

---

- Fast linear time algorithms
  - Construct Whole Program Stream representation
  - Detect hot data streams
- Few seconds, yet precise information



## Conclusions

---


- Cache-conscious layouts offer large benefits
- GC for cache-conscious data layouts
  - Structure layout reorganization
  - Hot/cold splitting
- Practical within context of CLR
  - Low-overhead data reference profiling
  - Fast, accurate data layout determination
- Opportunity for C# to approach performance of C/C++



## Future

---

- Implement and evaluate in the context of the CLR



## References

(<http://research.microsoft.com/~trishulc>)

---

- GC for cache-conscious data layout
  - Using GC to implement cache-conscious data layout, ISMM 98
  - Cache-conscious structure definition, PLDI 1999
- Low-overhead data reference profiling
  - Bursty tracing: A low-overhead framework for temporal profiling, FDDO 2001 submission
- Fast, accurate data layout determination
  - Efficient data reference locality abstractions and representations, PLDI 2001